طراحی و پیاده‌سازی زبان‌های برنامه‌سازی
Programming Languages: Design and Implementation

۴۰۳۶۴

نیمسال دوم ۱۴۰۰ – ۱۳۹۹

اهداف درس

- سیر طبیعی تحولات مفاهیم و روش‌های طراحی و پیاده‌سازی نسل‌های مختلف زبان‌های برنامه‌سازی به روشی تجربی و گام به گام
- مهندسی زبان‌های برنامه‌سازی (زبان‌های ویژه دامنه)
- آشنایی با پیاده سازی مفسرها به ویژه بر بستر ماشین‌های مجازی
- اصول طراحی زبان‌های برنامه‌سازی و روش‌ها و ساختارهای داده‌ای به کار رفته در پیاده‌سازی یا محقق کردن محیط برنامه نویسی
- ایجاد توانایی ارزیابی، مقایسه و انتخاب میان زبان‌های برنامه‌سازی موجود
- برنامه سازی تابعی
- معناشناسی و استدلال پیرامون برنامه

Grading and Other Admin affairs

- Your grade components:
  - Final Exam
  - Mid Term Exam
  - Assignments (paper based)
  - Assignments (programming including a final project)
  - Class Presentation …
- Class Administration Policies:

Main References

- Daniel P. Friedman and Mitchell Wand, ***Essentials of Programming Languages***, 3rd Edition, MIT Press, 2008.
- M.Felleisen, R.Findler, M.Flatt, S.Krishnamurthi, E.Barzilay, J.McCarthy, S.Tobin-Hochstadt, ***A Programmable Programming Language***, Communications of the ACM, Vol. 61, No. 3, Pp. 62-71, March 2018.
- https://racket-lang.org/
- My class and Slides

Other Useful General Text Books

- S.Krishnamurthi, ***Programming Languages: Application and Interpretation***, 2nd Edition, http://cs.brown.edu/courses/cs173/2012/book/book.pdf, 2017.

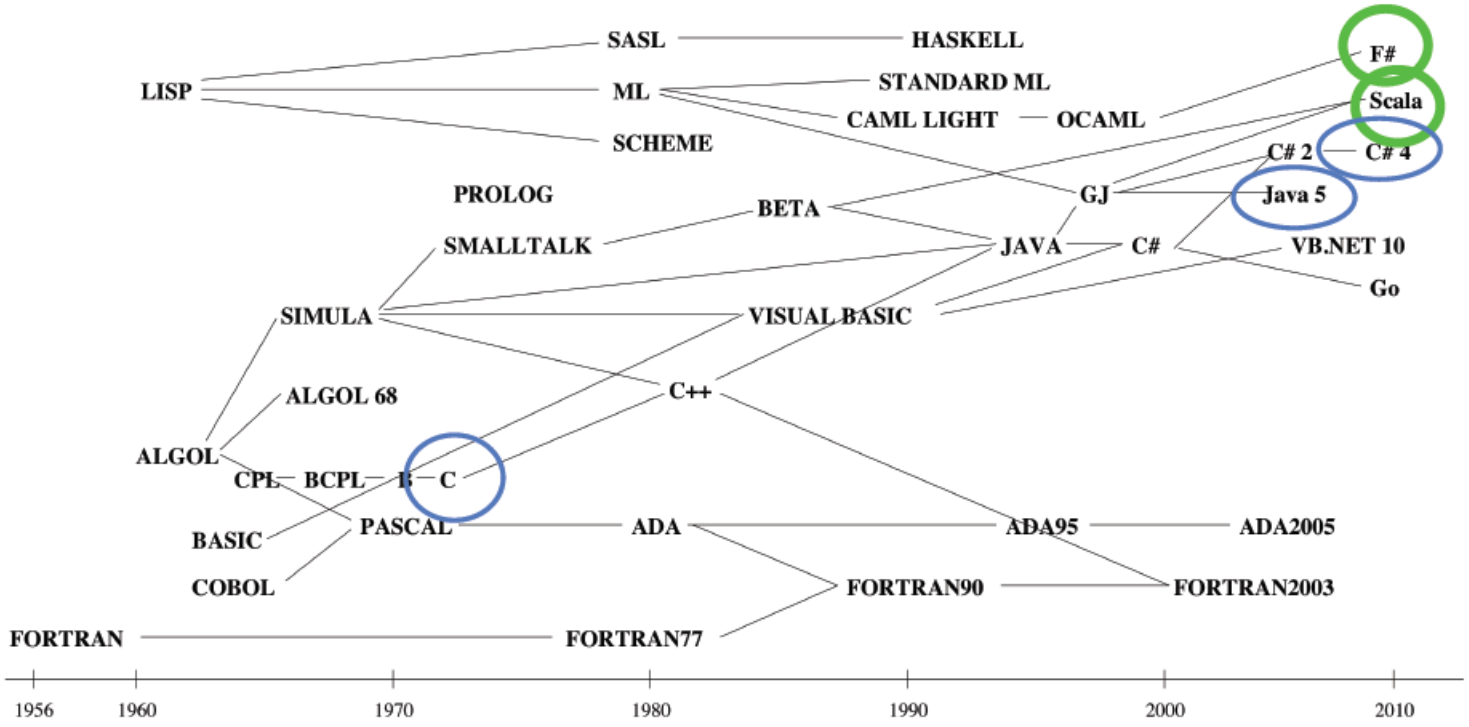
- Peter Sestoft, **Programming Language Concepts,** 2nd Edition, 2017.

- Michael Scott, **Programming Language Pragmatics**, (with its complimentary CD)

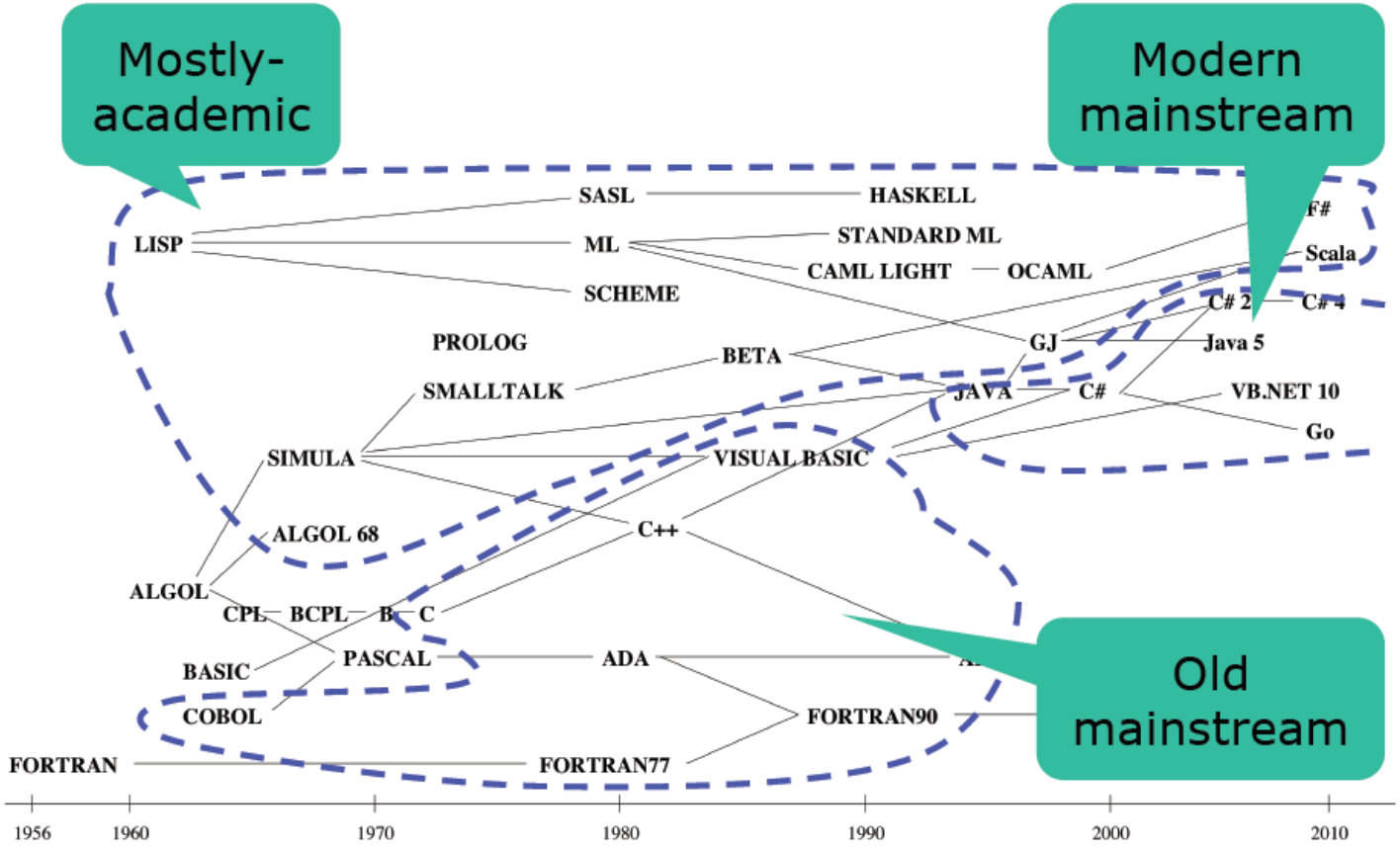
Other Useful General Text Books

- T.Pratt, M.Zelkowitz, **Programming Languages Design and Implementation**,
- B. MacLennan, **Principles of Programming Languages**
- C.Ghezzi, M.Jazayeri, **Programming Languages Concepts,**
- J.Mitchel, **Concepts in Programming Languages,**
- Ravi Sethi, **Programming Languages Concepts and Constructs,**

History of Programming Languages



History of Programming Languages



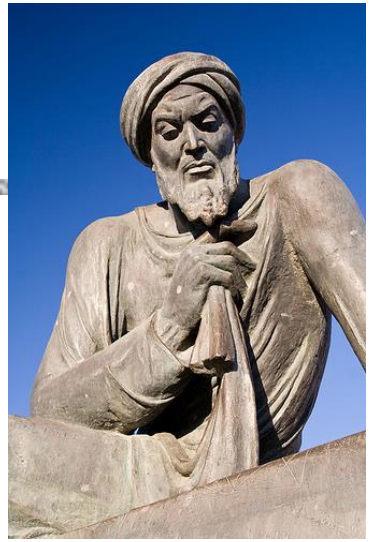
Potted History of Programming Languages

By: Vitaly Shmatikov

University of Texas, A&M

Cornell University

Algorithm



- Abu Ja'far Muhammad ibn Musa al-Khorezmi ("from Khorezm")
  - Lived in Baghdad around 780 – 850 AD
  - Chief mathematician in Khalif Al Mamun's "House of Wisdom"
  - Author of "A Compact Introduction To Calculation Using Rules Of Completion And Reduction"

Removing negative units from the equation by adding the same quantity on the other side ("al-gabr" in Arabic)

“Calculus of Thought”

- Gottfried Wilhelm Leibniz
  - 1646 - 1716
  - Inventor of calculus and binary system
  - “Calculus ratiocinator”: human reasoning can be reduced to a formal symbolic language, in which all arguments would be settled by mechanical manipulation of logical concepts
  - Invented a mechanical calculator

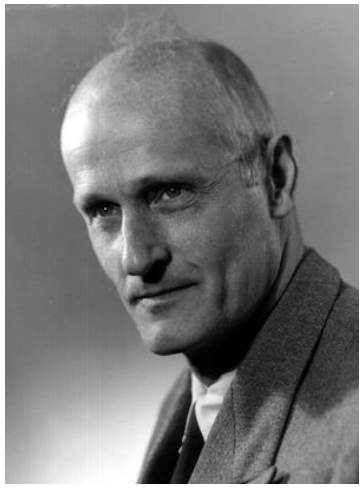
Formalisms for Computation (1)

- Predicate logic
  - Gottlöb Frege (1848-1925)
  - Formal basis for proof theory and automated theorem proving
  - Logic programming
    - Computation as logical deduction
- Turing machines
  - Alan Turing (1912-1954)
  - Imperative programming
    - Sequences of commands, explicit state transitions, update via assignment

Formalisms for Computation (2)

- Lambda calculus
  - Alonzo Church (1903-1995)
  - Formal basis for all functional languages, semantics, type theory
  - Functional programming
    - Pure expression evaluation, no assignment operator

- Recursive functions & automata
  - Stephen Kleene (1909-1994)
  - Regular expressions, finite-state machines, PDAs



Church's Thesis

- All these different syntactic formalisms describe the same class of mathematical objects
  - Church's Thesis: "Every effectively calculable function (effectively decidable predicate) is general recursive"
  - Turing's Thesis: "Every function which would be naturally regarded as computable is computable by a Turing machine"
- Recursion, lambda-calculus and Turing machines are equivalent in their expressive power
- Why is this a "thesis" and not a "theorem"?

Formalisms for Computation (3)

- Combinatory logic
  - Moses Schönfinkel (1889-1942??)
  - Haskell Curry (1900-1982)

- Post production systems
  - Emil Post (1897-1954)

- Markov algorithms
  - Andrey Markov (1903-1979)

Programming Language

- Formal notation for specifying computations
  - Syntax (usually specified by a context-free grammar)
  - Semantics for each syntactic construct
  - Practical implementation on a real or virtual machine
    - Translation vs. compilation vs. interpretation
      - C++ was originally translated into C by Stroustrup's Cfront
      - Java originally used a bytecode interpreter, now native code compilers are commonly used for greater efficiency
      - Lisp, Scheme and most other functional languages are interpreted by a virtual machine, but code is often precompiled to an internal executable for efficiency
    - Efficiency vs. portability

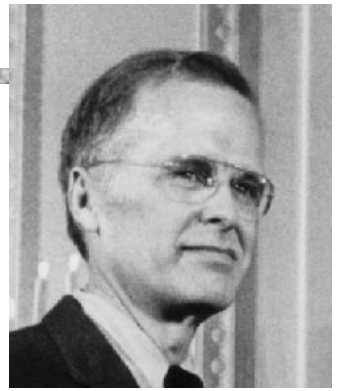
Assembly Languages

- Invented by machine designers the early 1950s
- Mnemonics instead of binary opcodes

```
push ebp
mov ebp, esp
sub esp, 4
push edi
```

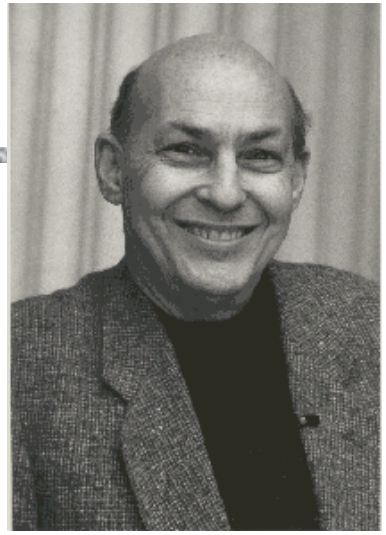
- Reusable macros and subroutines

FORTRAN



- Procedural, imperative language
  - Still used in scientific computation
- Developed at IBM in the 1950s by John Backus (1924-2007)
  - Backus's 1977 Turing award lecture (see course website) made the case for functional programming
  - On FORTRAN: "We did not know what we wanted and how to do it. It just sort of grew. The first struggle was over what the language would look like. Then how to parse expressions – it was a big problem…"
    - BNF: Backus-Naur form for defining context-free grammars

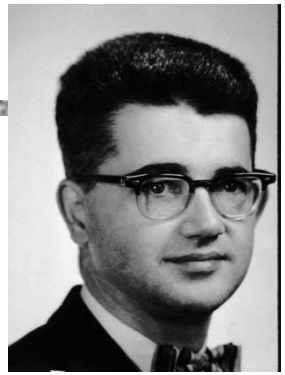
From FORTRAN to LISP

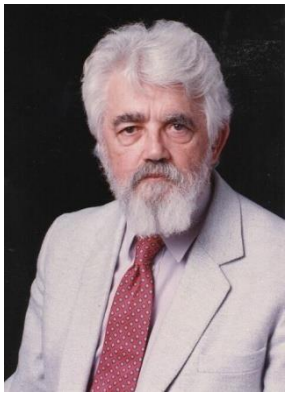


"Anyone could learn Lisp in one day, except that if they already knew FORTRAN, it would take three days"

- Marvin Minsky

LISP





- Invented by John McCarthy (b. 1927, Turing award: 1971)
  - See original paper on course website
- Formal notation for lambda-calculus
- Pioneered many PL concepts
  - Automated memory management (garbage collection)
  - Dynamic typing
  - No distinction between code and data
- Still in use: ACL2, Scheme, …

LISP Quotes

- “The greatest single programming language ever designed” --Alan Kay
- “LISP being the most powerful and cleanest of languages, that's the language that the GNU project always prefer” -- Richard Stallman
- “Programming in Lisp is like playing with the primordial forces of the universe. It feels like lightning between your fingertips.” -- Glenn Ehrlich
- “Lisp has all the visual appeal of oatmeal with fingernail clippings mixed in” -- Larry Wall
- “LISP programmers know the value of everything and the cost of nothing” -- Alan Perlis

Algol 60

- Designed in 1958-1960
- Great influence on modern languages
  - Formally specified syntax (BNF)
    - Peter Naur: 2005 Turing award
  - Lexical scoping: begin … end or {…}
  - Modular procedures, recursive procedures, variable type declarations, stack storage allocation

- "Birth of computer science" -- Dijkstra
- "A language so far ahead of its time that it was not only an improvement on its predecessors, but also on nearly all its successors" -- Hoare

Algol 60 Sample

```
real procedure average(A,n);
   real array A; integer n;
   begin
      real sum; sum := 0;
      for i = 1 step 1 until n do
            sum := sum + A[i];
      average := sum/n
   end;
```

no array bounds

no ; here

set procedure return value by assignment

Algol Oddity

- Question
  - Is x := x equivalent to doing nothing?
- Interesting answer in Algol

```
integer procedure p;
begin
        ...
        p := p
        ...
  end;
```

  - Assignment here is actually a recursive call

Some Trouble Spots in Algol 60

- Type discipline improved by later languages
  - Parameter types can be array
    - No array bounds
  - Parameter type can be procedure
    - No argument or return types for procedure parameter
- Parameter passing methods
  - Pass-by-name had various anomalies
    - "Copy rule" based on substitution, interacts with side effects
  - Pass-by-value expensive for arrays
- Some awkward control issues
  - Goto out of block requires memory management

Algol 60 Pass-by-Name

- Substitute text of actual parameter
  - Unpredictable with side effects!
- Example

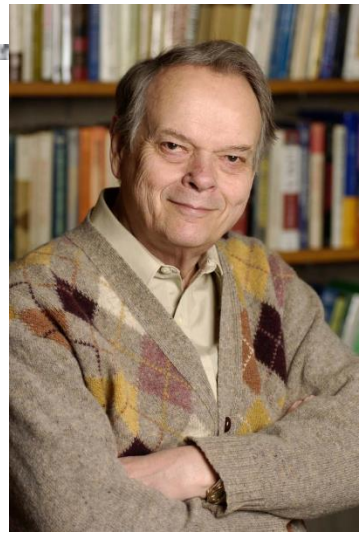
```
procedure inc2(i, j);
   integer i, j;
   begin
      i := i+1;
      j := j+1
   end;
inc2 (k, A[k]);
```

→

```
begin
   k := k+1;
   A[k] := A[k] +1
end;
```

Is this what you expected?

Algol 60 Legacy



"Another line of development stemming from Algol 60 has led to languages such as Pascal and its descendants, e.g., Euclid, Mesa, and Ada, which are significantly lowerlevel than Algol. Each of these languages seriously restricts the block or procedure mechanism of Algol by eliminating features such as call by name, dynamic arrays, or procedure parameters."

- John C. Reynolds

Algol 68

- Very elaborate type system
  - Complicated type conversions
  - Idiosyncratic terminology
    - Types were called "modes"
    - Arrays were called "multiple values"
- vW grammars instead of BNF
  - Context-sensitive grammar invented by A. van Wijngaarden
- Eliminated pass-by-name
- Considered difficult to understand

Pascal

- Designed by Niklaus Wirth
  - 1984 Turing Award
- Revised type system of Algol
  - Good data structure concepts
    - Records, variants, subranges
  - More restrictive than Algol 60/68
    - Procedure parameters cannot have procedure parameters
- Popular teaching language
- Simple one-pass compiler

Limitations of Pascal

- Array bounds part of type

  illegal

  procedure p(a: array [1..10] of integer)

  procedure p(n: integer, a: array [1..n] of integer)

  - Attempt at orthogonal design backfires
    - Parameter must be given a type
    - Type cannot contain variables

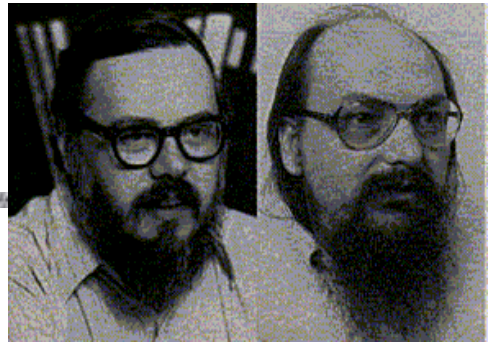
  How could this have happened? Emphasis on teaching!

- Not successful for "industrial-strength" projects
  - See Kernighan's "Why Pascal is not my favorite language" on the course website

SIMULA 67

- Ole-Johan Dahl (1931-2002)
- Kristen Nygaard (1926-2002)
  - Joint 2001 Turing Award
- First <u>object-oriented</u> language
  - Objects and classes
  - Subclasses and inheritance
  - Virtual procedures

BCPL / B / C Family



- Born of frustration with big OSes and big languages (Multics, PL/I, Algol 68)
- Keep lexical scope and recursion
- Low-level machine access
  - Manual memory management
  - Explicit pointer manipulation
  - Weak typing (introduced in C)
- Systems programming for small-memory machines
  - PDP-7, PDP-11, later VAX, Unix workstations and PCs
  - C has been called a "portable assembly language"

BCPL

- Designed by Martin Richards (1966)
- Emphasis on portability and ease of compilation
  - Front end: parse + generate code for virtual machine
  - Back end: translate code for native machine
- Single data type (word), equivalence of pointers and arrays, pointer arithmetic – this is unusual!

"The philosophy of BCPL is not one of the tyrant who thinks he knows best and lays down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, even when confronted with apparent nonsense. The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions."

Arrays and Pointers

- An array is treated as a pointer to first element
- BCPL: let V = vec 10

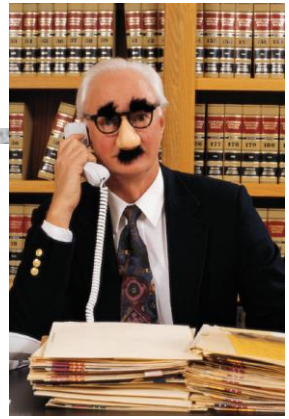
  V!i to index the ith array element
- C: A[i] is equivalent to

  pointer dereference *((A) + (i))

B

- "BCPL squeezed into 8K bytes of memory & filtered through Ken Thompson's brain"
- Very compact syntax
  - One-pass compiler on a small-memory machine
    - Generates intermediate "threaded code," not native code
  - No nested scopes
  - Assignment: = instead of Algol-style :=
    - How many times have you written if (a=b) { … } ?
  - Pre-/postfix notation: x++ instead of x:=x+1
  - Null-terminated strings
    - In C, strings are null-terminated sequences of bytes referenced either by a pointer-to-char, or an array variable s[]

Lex the Language Lawyer



Can only be applied to l-value (more about this later in the course)

++x++

This is evaluated first

Increments x, returns old value

Not an l-value! This is illegal in C!

Now C++ …

```
class DoublePlus {
public:
    // prefix operator
    DoublePlus operator++() { … }
    // postfix operator
    DoublePlus operator++(int) { … }
};
```

What is this for?

More Fun with Prefix and Postfix

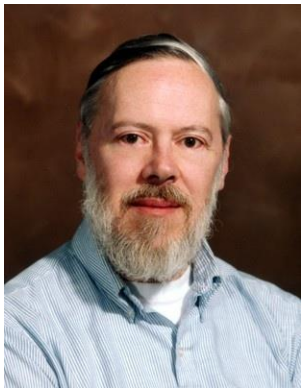
What do these mean?

x+=x++

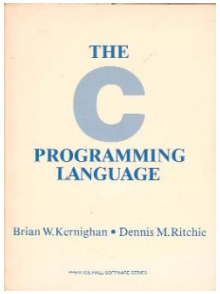
++x + x++

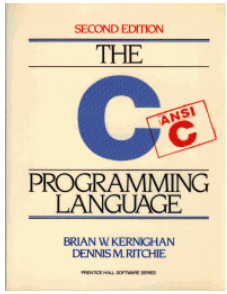
C

- Bell Labs 1972 (Dennis Ritchie)
- Development closely related to UNIX
  - 1983 Turing Award to Thompson and Ritchie



- Added weak typing to B
  - int, char, their pointer types
  - Typed arrays = typed pointers
    - int a[10]; … x = a[i]; means
      x = *(&a[0]+i*sizeof(int))
- Compiles to native code





Types in C

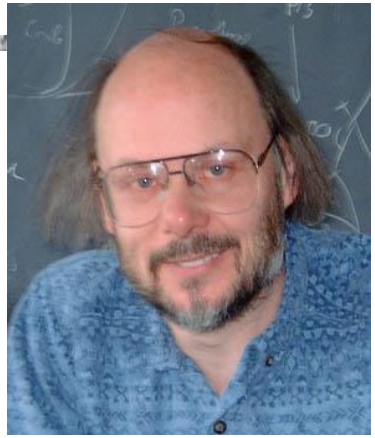
- Main difference between B and C
- Syntax of type rules influenced by Algol 68
  - int i, *pi, **ppi;
  - int f(), *f(), **f(), *(*pf)(), (*pf)(int);
  - int *api[10], (*pai)[10];
- Also structs and unions

What do these declarations mean?

Evolution of C

- 1973-1980: new features; compiler ported
  - unsigned, long, union, enums
- 1978: K&R C book published
- 1989: ANSI C standardization
  - Function prototypes as in C++
- 1999: ISO 9899:1999 also known as “C99”
  - Inline functions, C++-like decls, bools, variable arrays
- Concurrent C, Objective C, C*, C++, C#
- “Portable assembly language”
  - Early C++, Modula-3, Eiffel source-translated to C

C++



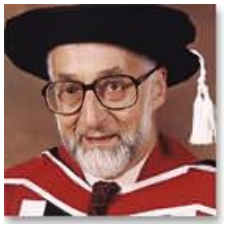
- Bell Labs 1979 (Bjarne Stroustrup)
  - "C with Classes" (C++ since 1983)
- Influenced by Simula
- Originally translated into C using Cfront, then native compilers
  - GNU g++
- Several PL concepts
  - Multiple inheritance
  - Templates / generics
  - Exception handling

Java

- Sun 1991-1995 (James Gosling)
  - Originally called Oak, intended for set top boxes
- Mixture of C and Modula-3
  - Unlike C++
    - No templates (generics), no multiple inheritance, no operator overloading
  - Like Modula-3 (developed at DEC SRC)
    - Explicit interfaces, single inheritance, exception handling, built-in threading model, references & automatic garbage collection (no explicit pointers!)
- "Generics" added later

Other Important Languages

- Algol-like
  - Modula, Oberon, Ada
- Functional
  - ISWIM, FP, SASL, Miranda, Haskell, LCF, ML, Caml, Ocaml, Scheme, Common LISP
- Object-oriented
  - Smalltalk, Objective-C, Eiffel, Modula-3, Self, C#, CLOS
- Logic programming
  - Prolog, Gödel, LDL, ACL2, Isabelle, HOL



… And More

- Data processing and databases
  - Cobol, SQL, 4GLs, XQuery
- Systems programming
  - PL/I, PL/M, BLISS
- Specialized applications
  - APL, Forth, Icon, Logo, SNOBOL4, GPSS, Visual Basic
- Concurrent, parallel, distributed
  - Concurrent Pascal, Concurrent C, C*, SR, Occam, Erlang, Obliq

Forth

- Program BIOS, bootloaders, device firmware
  - Sun BIOS, Lockheed Martin's missile tracking, FedEx barcode readers …

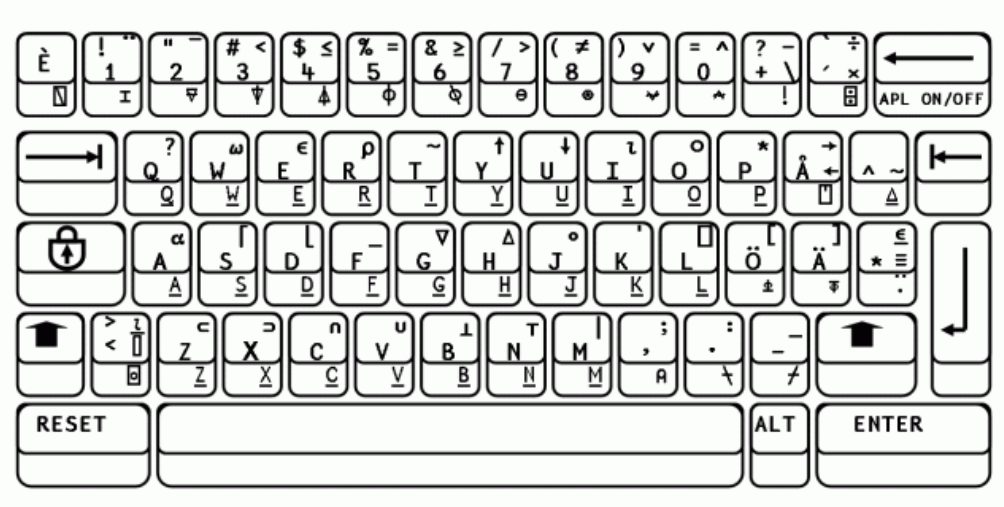
```
hex 4666 dup negate do i 4000 dup 2* negate
  do 2a 0 dup 2dup 1e 0 do 2swap * d >>a 4
  pick + -rot - j + dup dup * e >>a rot dup
  dup * e >>a rot swap 2dup + 10000 > if
  3drop 2drop 20 0 dup 2dup leave then loop
  2drop 2drop type 268 +loop cr drop 5de
  +loop
```

APL

- Computation-intensive tasks, esp. in finance
  - Mortgage cash flow analysis, insurance calculations, …

Got this?

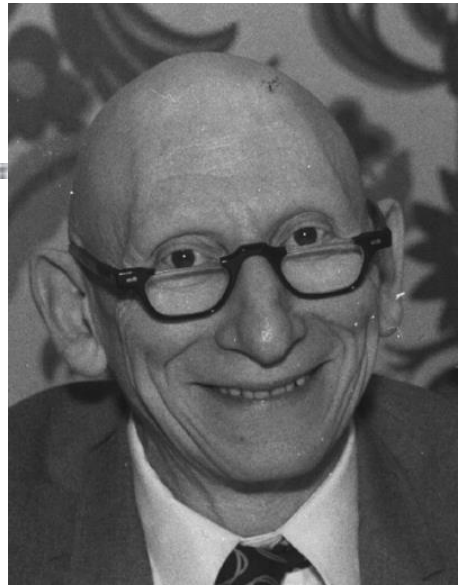
```
[6]     L←(L⍳':')↓L←,L
[7]     L←LJUST ⍕TOM',',L
[8]     S←¯1++/∧\L≠'('
[9]     X←0⌈⌈/S
[10]    L←S⌽(-(⍴L)+0,X)↑L
[11]    A←((1↑⍴L),X)↑L
[12]    N←0 1↓DLTB(0,X)↓L
[13]    N←,'α',N
[14]    N[(N='_')/⍳⍴N]←' '
[15]    N←0 ¯1↓RJUST ⍕TOM N
[16]    S←+/∧\' '≠⌽N
```



Brave New World

- Programming tool "mini-languages"
  - awk, make, lex, yacc, autoconf …
- Command shells, scripting and "web" languages
  - sh, csh, tcsh, ksh, zsh, bash …
  - Perl, JavaScript, PHP, Python, Rexx, Ruby, Tcl, AppleScript, VBScript …
- Web application frameworks and technologies
  - ASP.NET, AJAX, Flash, Silverlight …
    - **Note:** HTML/XML are markup languages, <u>not</u> programming languages, but they often embed executable scripts like Active Server Pages (ASPs) & Java Server Pages (JSPs)

Why So Many Languages?



"There will always be things we wish to say in our programs that in all languages can only be said poorly."

- Alan Perlis

What's Driving Their Evolution?

- Constant search for better ways to build software tools for solving computational problems
  - Many PLs are general purpose tools
  - Others are targeted at specific kinds of problems
    - For example, massively parallel computations or graphics
- Useful ideas evolve into language designs
  - Algol → Simula → Smalltalk → C with Classes → C++
- Often design is driven by expediency
  - Scripting languages: Perl, Tcl, Python, PHP, etc.
    - "PHP is a minor evil perpetrated by incompetent amateurs, whereas Perl is a great and insidious evil, perpetrated by skilled but perverted professionals." - Jon Ribbens

What Do They Have in Common?

- Lexical structure and analysis
  - Tokens: keywords, operators, symbols, variables
  - Regular expressions and finite automata
- Syntactic structure and analysis
  - Parsing, context-free grammars
- Pragmatic issues
  - Scoping, block structure, local variables
  - Procedures, parameter passing, iteration, recursion
  - Type checking, data structures
- Semantics
  - What do programs mean and are they correct

Core Features vs. Syntactic Sugar

- What is the core high-level language syntax required to emulate a universal Turing machine?
  - What is the core syntax of C?
    - Are ++, --, +=, -=, ?:, for/do/while part of the core?
- Convenience features?
  - Structures/records, arrays, loops, case/switch?
  - Preprocessor macros (textual substitution)
  - Run-time libraries
    - String handling, I/O, system calls, threads, networking, etc.
  - "Syntactic sugar causes cancer of the semicolons"

    - Alan Perlis

Final Thoughts

- There will be new languages invented
  - You will have to spend time learning them on your own!
  - For now, enjoy the luxury of being able to take a class
- Conflicting goals for language design can lead to feature creep and hideous complexity
  - Exhibit A: PL/I
  - Exhibit B: C++
- Then someone gets fed up …
  - A language that adopts the original simple and elegant ideas, while eliminating the complexity (e.g., Java)

Programming Languages: Design and Implementation

Translators and Interpreters

CE 40364

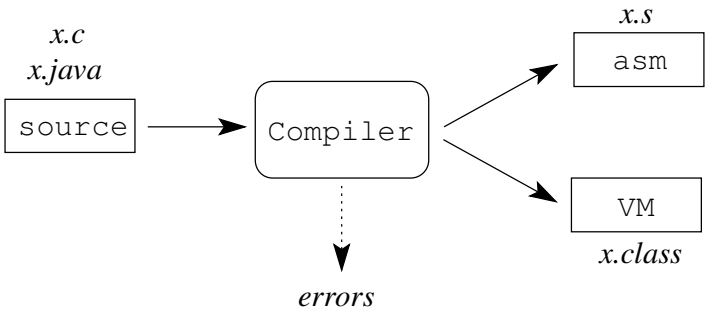
Fall1399-1400

Session 2

TRANSLATORS

What's a Compiler???

- At the very basic level a compiler translates a computer program from source code to some kind of executable code:



- Often the source code is simply a text file and the executable code is a resulting assembly language program: `gcc -S x.c` reads the C source file `x.c` and generates an assembly code file `x.s`. Or the output can be a virtual machine code: `javac x.java` produces `x.class`.

What's a Language Translator???

- A compiler is really a special case of a language translator.
- A translator is a program that transforms a "program" P_1 written in a language L_1 into a program P_2 written in another language L_2.
- Typically, we desire P_1 and P_2 to be semantically equivalent, i.e. they should behave identically.

Example Language Translators

source language	translator	target language
LaTeX	$\xrightarrow{\texttt{latex2html}}$	html
Postscript	$\xrightarrow{\texttt{ps2ascii}}$	text
FORTRAN	$\xrightarrow{\texttt{f2c}}$	C
C++	$\xrightarrow{\texttt{cfront}}$	C
C	$\xrightarrow{\texttt{gcc}}$	assembly
.class	$\xrightarrow{\texttt{SourceAgain}}$	Java
x86 binary	$\xrightarrow{\texttt{fx32}}$	Alpha binary

Compiler Input

Text File Common on Unix.

Syntax Tree A structure editor uses its knowledge of the source language syntax to help the user edit & run the program. It can send a syntax tree to the compiler, relieving it of lexing & parsing.

Compiler Output

Assembly Code Unix compilers do this. Slow, but easy for the compiler.

Object Code .o-files on Unix. Faster, since we don't have to call the assembler.

Executable Code Called a load-and-go-compiler.

Abstract Machine Code Serves as input to an interpreter. Fast turnaround time.

C-code Good for portability.

Compiler Tasks

Static Semantic Analysis Is the program (statically) correct? If not, produce error messages to the user.

Code Generation The compiler must produce code that can be executed.

Symbolic Debug Information The compiler should produce a description of the source program needed by symbolic debuggers. Try `man gdb`.

Cross References The compiler may produce **cross-referencing** information. Where are identifiers declared & referenced?

Profiler Information Where does my program spend most of its execution time? Try `man gprof`.

Compiler Phases

ANALYSIS	SYNTHESIS
Lexical Analysis	Intermediate Code Generation
Syntactic Analysis	Code Optimization
Semantic Analysis	Machine Code Generation

Multi-pass Compilation

- The next slide shows the outline of a typical compiler. In a unix environment each pass could be a stand-alone program, and the passes could be connected by pipes:

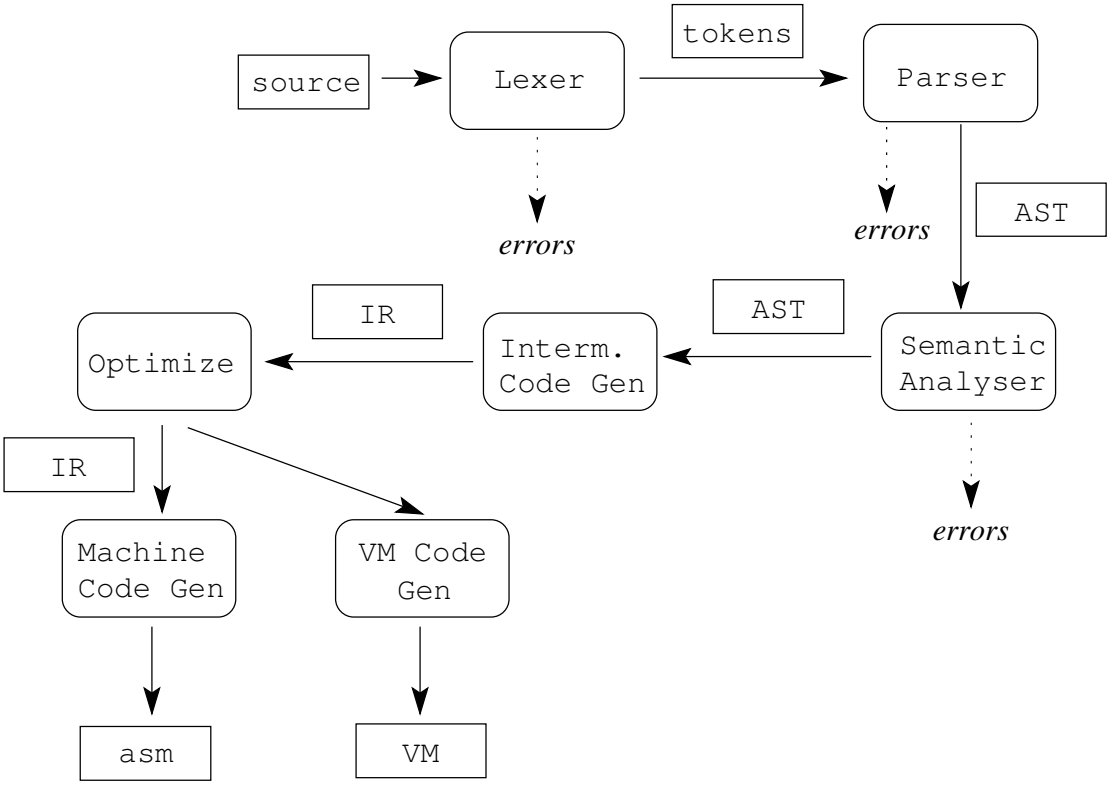
```
lex x.c | parse | sem | ir | opt | codegen > x.s
```

- For performance reasons the passes are usually integrated:

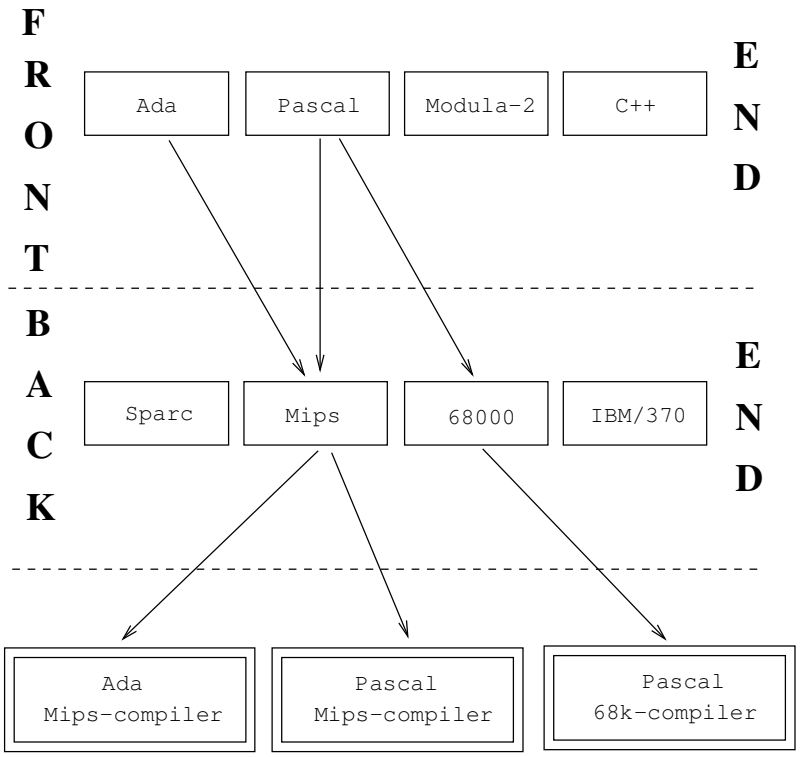
```
front x.c > x.ir
back x.ir > x.s
```

The front-end does all analysis and IR generation. The back-end optimizes and generates code.

Multi-pass Compilation…



Mix-and-Match Compilers



INTERPRETERS

Interpretation

- An interpreter is like a CPU, only in software.
- The compiler generates *virtual machine* (VM) code rather than native machine code.
- The interpreter executes VM instructions rather than native machine code.

Interpreters are

slow Often 10–100 times slower than executing machine code directly.

portable The virtual machine code is not tied to any particular architecture.

Interpretation…

Interpreters are

slow Often 10–100 times slower than executing machine code directly.

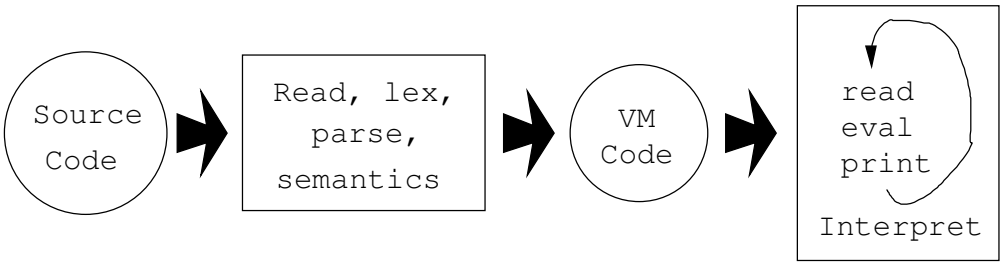
portable The virtual machine code is not tied to any particular architecture.

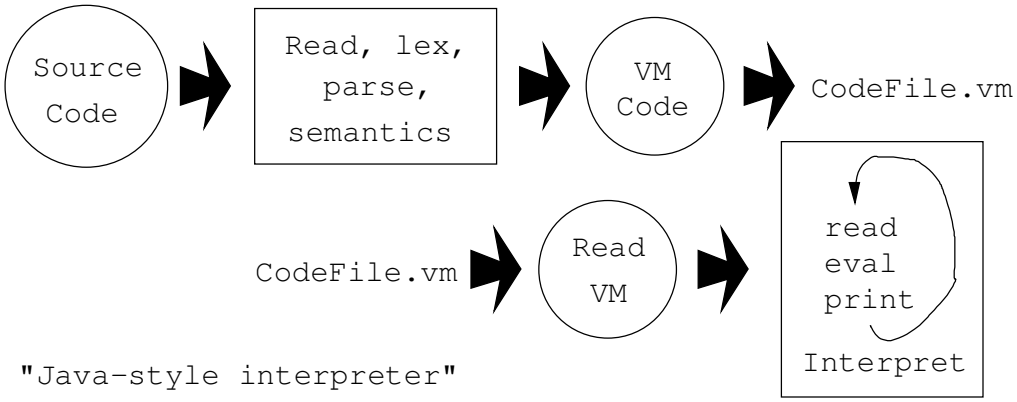
Interpreters work well with

very high-level, dynamic languages (APL,Prolog,ICON) where a lot is unknown at compile-time (array bounds, etc).

Kinds of Interpreters

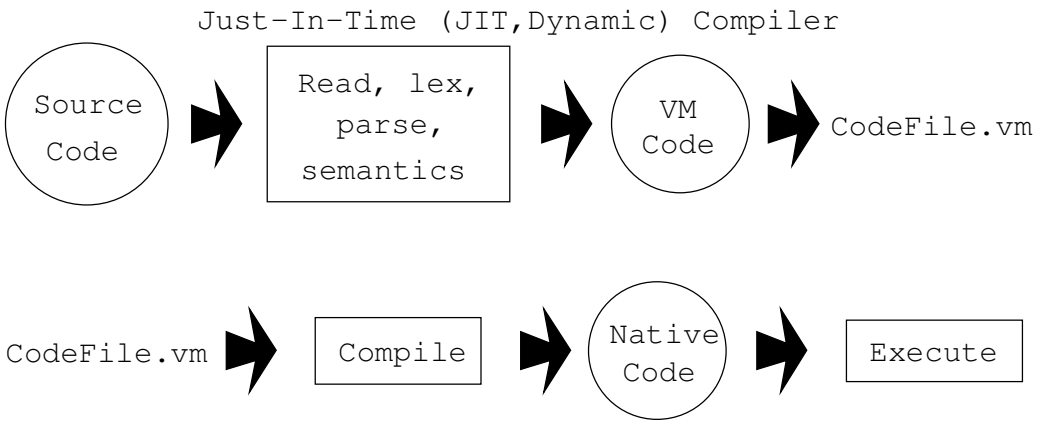
"APL/Prolog-style (load-and-go/interactive) interpreter"





"Java-style interpreter"

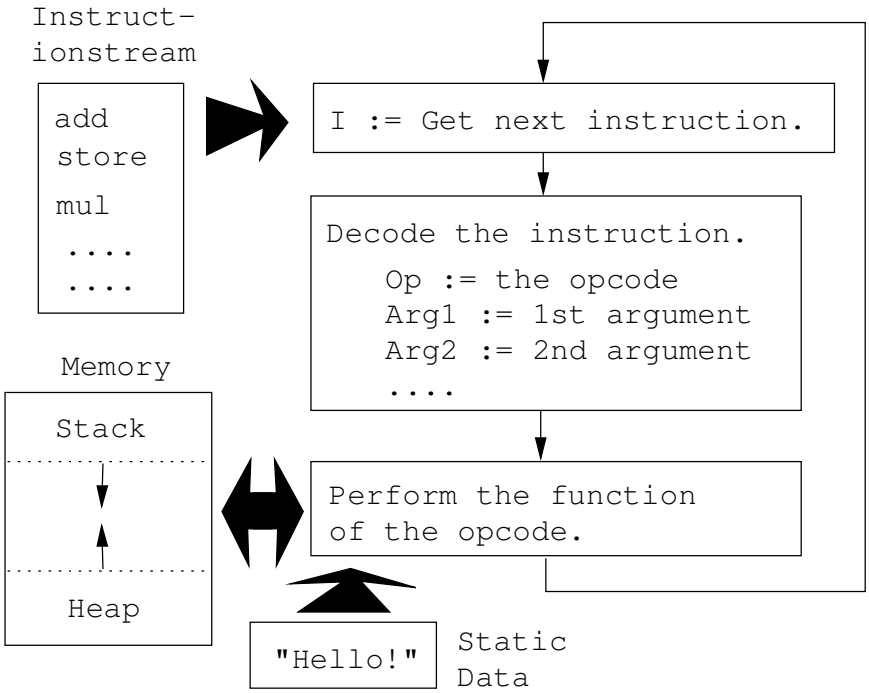
Kinds of Interpreters…



Actions in an Interpreter

- Internally, an interpreter consists of
  1. The interpreter *engine*, which executes the VM instructions.
  2. *Memory* for storing user data. Often separated as a heap and a stack.
  3. A stream of VM instructions.

Actions in an Interpreter…



Stack-Based Instruction Sets

- Many virtual machine instruction sets (e.g. Java bytecode, Forth) are *stack based*.

  add pop the two top elements off the stack, add them together, and push the result on the stack.

  push X push the value of variable X.

  pusha X push the address of variable X.

  store pop a value V, and an address A off the stack. Store V at memory address A.

Stack-Based Instruction Sets…

- Here's an example of a small program and the corresponding stack code:

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`pusha X`
`BEGIN`	`push Y`
`  X := Y + Z;`	`push Z`
`END;`	`add`
	`store`

Register-Based Instruction Sets

- Stack codes are *compact*. If we don't worry about code size, we can use any intermediate code (tuples, trees). Example: RISC-like VM code with ∞ number of virtual registers $R_1, \cdots$:

add R_1, R_2, R_3 Add VM registers R_2 and R_3 and store in VM register R_1.

load R_1, X R_1 := value of variable X.

loada R_1, X R_1 := address of variable X.

store R_1, R_2 Store value R_2 at address R_1.

Register-Based Instruction Sets…

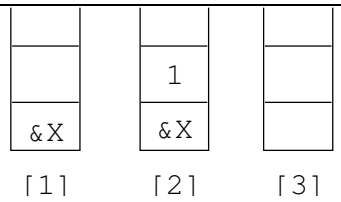
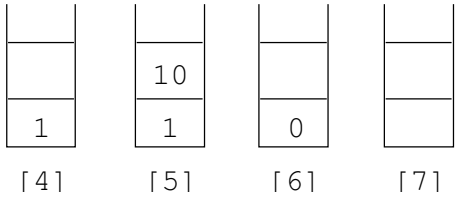
- Here's an example of a small program and the corresponding register code:

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	load R_1, Y
`BEGIN`	load R_2, Z
`  X := Y + Z;`	add R_3, R_1, R_2
`END;`	loada R_4, X
	store R_4, R_3

Stack Machine Example I

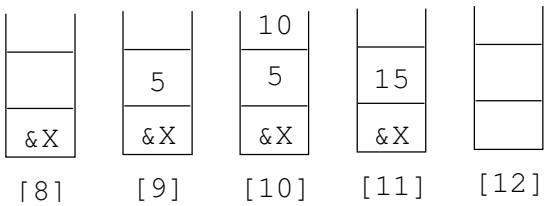
Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`[1] pusha X`
`BEGIN`	`[2] push 1`
`  X := 1;`	`[3] store`
`  WHILE X < 10 DO`	`[4] push X`
	`[5] push 10`
	`[6] GE`
	`[7] BrTrue 14`
`    X := Y + Z;`	`[8] pusha X`
	`[9] push Y`
	`[10] push Z`
	`[11] add`
`  ENDDO`	`[12] store`
`END.`	`[13] dump 4`

Stack Machine Example (a)

VM Code	Stack	Memory
[1] pusha X [2] push 1 [3] store		X 1 Y 5 Z 10
[4] push X [5] push 10 [6] GE [7] BrTrue 14		X 1 Y 5 Z 10

Stack Machine Example (b)

VM Code	Stack	Memory

`[8] pusha X`
`[9] push Y`
`[10] push Z`
`[11] add`
`[12] store`
		X 15
		Y 5
		Z 10

`[13] jump 4`		

Switch Threading

Switch Threading

- Instructions are stored as an array of integer tokens. A switch selects the right code for each instruction.

```
typedef enum {add,load,store,···} Inst;
void engine () {
   static Inst prog[] = {load,add,···};
   Inst *pc = &prog;
   int Stack[100]; int sp = 0;
   for (;;)
     switch (*pc++) {
       case add:  Stack[sp-1]=Stack[sp-1]+Stack[sp];
                  sp--; break;
   }}}
```

Switch Threading in Java

- Let's look at a simple Java switch interpreter.
- We have a stack of integers `stack` and a stack pointer `sp`.
- There's an array of bytecodes `prog` and a program counter `pc`.
- There is a small memory area `memory`, an array of 256 integers, numbered 0–255. The `LOAD`, `STORE`, `ALOAD`, and `ASTORE` instructions access these memory cells.

Bytecode semantics

mnemonic	opcode	stack-pre	stack-post	side-effects
ADD	0	[A,B]	[A+B]	
SUB	1	[A,B]	[A-B]	
MUL	2	[A,B]	[A*B]	
DIV	3	[A,B]	[A-B]	
LOAD X	4	[]	[Memory[X]]	
STORE X	5	[A]	[]	Memory[X] = A
PUSHB X	6	[]	[X]	
PRINT	7	[A]	[]	Print A
PRINTLN	8	[]	[]	Print a newline
EXIT	9	[]	[]	The interpreter exits
PUSHW X	11	[]	[X]	

Bytecode semantics…

mnemonic	opcode	stack-pre	stack-post	side-effects
BEQ L	12	[A,B]	[]	if A=B then PC+=L
BNE L	13	[A,B]	[]	if A!=B then PC+=L
BLT L	14	[A,B]	[]	if A<B then PC+=L
BGT L	15	[A,B]	[]	if A>B then PC+=L
BLE L	16	[A,B]	[]	if A<=B then PC+=L
BGE L	17	[A,B]	[]	if A>=B then PC+=L
BRA L	18	[]	[]	PC+=L
ALOAD	19	[X]	[Memory[X]]	
ASTORE	20	[A,X]	[]	Memory[X] = A
SWAP	21	[A,B]	[B,A]	

Example programs

This program prints a newline character and then exits:

```
PRINTLN
EXIT
```

Or, in binary: $\langle 8, 9 \rangle$

This program prints the number 10, then a newline character, and then exits:

```
PUSHB 10
PRINT
PRINTLN
EXIT
```

Or, in binary: $\langle 6, 10, 7, 8, 9 \rangle$

Example programs…

This program pushes two values on the stack, then performs an `ADD` instruction which pops these two values off the stack, adds them, and pushes the result. `PRINT` then pops this value off the stack and prints it:

```
PUSHB 10
PUSHB 20
ADD
PRINT
PRINTLN
EXIT
```

Or, in binary: $\langle 6, 10, 6, 20, 0, 7, 8, 9 \rangle$

Example program…

This program uses the `LOAD` and `STORE` instructions to store a value in memory cell number 7:

```
PUSHB 10
STORE 7
PUSHB 10
LOAD 7
MUL
PRINT
PRINTLN
EXIT
```

Or, in binary: $\langle 6, 10, 5, 7, 6, 10, 4, 7, 2, 7, 8, 9 \rangle$

Example programs…

```
# Print the numbers 1 through 9.
# i = 1; while (i < 10) do {print i; println; i++;}
PUSHB 1       # mem[1] = 1;
STORE 1
LOAD 1        # if mem[1] < 10 goto exit
PUSHB 10
BGE
LOAD 1        # print mem[i] value
PRINT
PRINTLN
PUSHB 1       # mem[1]++
LOAD 1
ADD
STORE 1
BRA           # goto top of loop
EXIT
```

Bytecode Description

`ADD`: Pop the two top integers A and B off the stack, then push $A + B$.

`SUB`: As above, but push $A - B$.

`MUL`: As above, but push $A * B$.

`DIV`: As above, but push A/B.

`PUSHB` X: Push X, a signed, byte-size, value, on the stack.

`PUSHW` X: Push X, a signed, word-size, value, on the stack.

`PRINT`: Pop the top integer off the stack and print it.

`PRINTLN`: Print a newline character.

`EXIT`: Exit the interpreter.

Bytecode Description…

`LOAD` X: Push the contents of memory cell number X on the stack.

`STORE` X: Pop the top integer off the stack and store this value in memory cell number X.

`ALOAD`: Pop the address of memory cell number X off the stack and push the value of X.

`ASTORE`: Pop the address of memory cell number X and the value V off the stack and store the V in X.

`SWAP`: Exchange the two top elements on the stack.

Bytecode Description...

$\boxed{\texttt{BEQ}\ L}$: Pop the two top integers A and B off the stack, if $A == B$ then continue with instruction $\texttt{PC} + L$, where $\texttt{PC}$ is address of the instruction *following* this one. Otherwise, continue with the next instruction.

$\boxed{\texttt{BNE}\ L}$: As above, but branch if $A \neq B$.

$\boxed{\texttt{BLT}\ L}$: As above, but branch if $A < B$.

$\boxed{\texttt{BGT}\ L}$: As above, but branch if $A > B$.

$\boxed{\texttt{BLE}\ L}$: As above, but branch if $A \leq B$.

$\boxed{\texttt{BGE}\ L}$: As above, but branch if $A \geq B$.

$\boxed{\texttt{BRA}\ L}$: Continue with instruction $\texttt{PC} + L$, where $\texttt{PC}$ is the address of the instruction *following* this one.

Switch Threading in Java

```
public class Interpreter {
   static final byte ADD    = 0;
   static final byte SUB    = 1;
   static final byte MUL    = 2;
   static final byte DIV    = 3;
   static final byte LOAD   = 4;
   static final byte STORE  = 5;
   static final byte PUSHB  = 6;
   static final byte PRINT  = 7;
   static final byte PRINTLN= 8;
   static final byte EXIT   = 9;
   static final byte PUSHW  = 11;
```

```
static final byte BEQ    = 12;
static final byte BNE    = 13;
static final byte BLT    = 14;
static final byte BGT    = 15;
static final byte BLE    = 16;
static final byte BGE    = 17;
static final byte BRA    = 18;
static final byte ALOAD  = 19;
static final byte ASTORE = 20;
static final byte SWAP   = 21;
```

```
static void interpret (byte[] prog)
       throws Exception {
    int[] stack = new int[100];
    int[] memory = new int[256];
    int pc = 0;
    int sp = 0;
    while (true) {
    switch (prog[pc]) {
       case ADD    : {
           stack[sp-2]+=stack[sp-1]; sp--;
           pc++; break;
         }
       /* Same for SUB, MUL, DIV. */
```

```
case LOAD   : {
  stack[sp] = memory[(int)prog[pc+1]];
  sp++; pc+=2; break;}

case STORE  : {
  memory[prog[pc+1]] = stack[sp-1];
  sp-=1; pc+=2; break;}

case ALOAD   : {
  stack[sp-1] = memory[stack[sp-1]];
  pc++; break;}

 case ASTORE  : {
   memory[stack[sp-1]] = stack[sp-2];
   sp-=2; pc++; break;}
```

```
case SWAP : {
   int tmp = stack[sp-1];
   stack[sp-1] = stack[sp-2];
   stack[sp-2]=tmp;
   pc++; break; }

 case PUSHB  : {
   stack[sp] = (int)prog[pc+1];
   sp++; pc+=2; break; }
 /* Similar for PUSHW. */

 case PRINT  : {
   System.out.print(stack[--sp]);
   pc++; break; }
```

```
        case PRINTLN: {
          System.out.println(); pc++; break; }

        case EXIT : {return;}

        case BEQ  : {/*Same for BNE,BLT,...*/
          pc+= (stack[sp-2]==stack[sp-1])?
                    2+(int)prog[pc+1]:2;
          sp-=2; break; }

        case BRA  : {
           pc+= 2+(int)prog[pc+1]; break; }

        default : throw new Exception("Illegal")
}}}}
```

Switch Threading…

- Switch (case) statements are implemented as indirect jumps through an array of label addresses (a *jump-table*). Every switch does 1 range check, 1 table lookup, and 1 jump.

```
switch (e) {
 case 1:  S1; break;
 case 3:  S2; break;
 default:  S3;
}
```

$\Rightarrow$

```
JumpTab = {0,&Lab1,&Lab3,&Lab2};
if ((e < 1) || (e > 3)) goto Lab3;
goto *JumpTab[e];
Lab1:  S1; goto Lab4;
Lab2:  S2; goto Lab4;
Lab3:  S3;
Lab4:
```

Faster Operator Dispatch

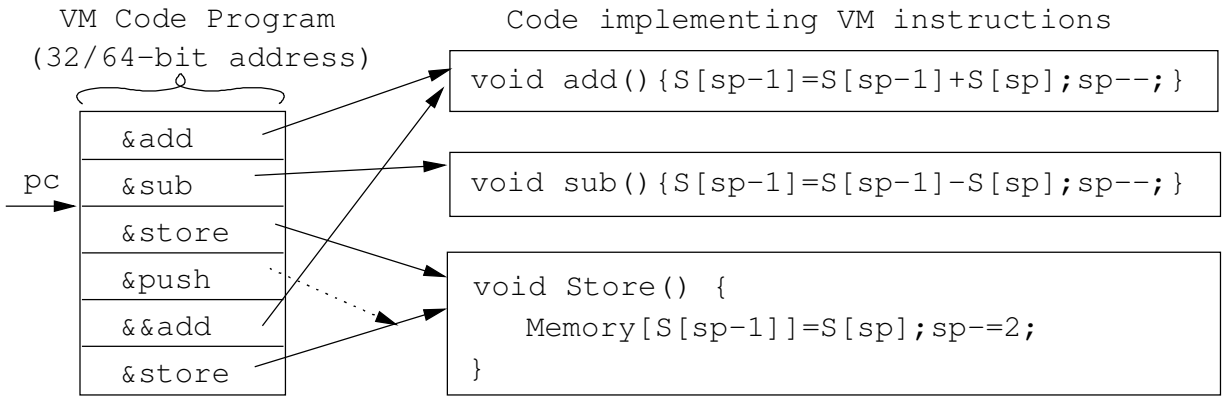
Direct Call Threading

- Every instruction is a separate function.
- The program `prog` is an array of pointers to these functions.
- I.e. the `add` instruction is represented as the address of the `add` function.
- `pc` is a pointer to the current instruction in `prog`.
- `(*pc++)()` jumps to the function that `pc` points to, then increments `pc` to point to the next instruction.
- Hard to implement in Java.

Direct Call Threading…

```
typedef void (* Inst)();
Inst prog[] = {&load,&add,···};

Inst *pc = &prog;
int Stack[100]; int sp = 0;

void add(); {
   Stack[sp-1]=Stack[sp-1]+Stack[sp];
   sp--;}

void engine () {
   for (;;) (*pc++)()
}
```

Direct Call Threading…



Direct Call Threading…

- In direct call threading all instructions are in their own functions.
- This means that VM registers (such as `pc`, `sp`) must be in global variables.
- So, every time we access `pc` or `sp` we have to load them from global memory. $\Rightarrow$ Slow.
- With the switch method `pc` and `sp` are local variables. Most compilers will keep them in registers. $\Rightarrow$ Faster.
- Also, a direct call threaded program will be large since each instruction is represented as a 32/64-bit address.
- Also, overhead from call/return sequence.

Direct Threading

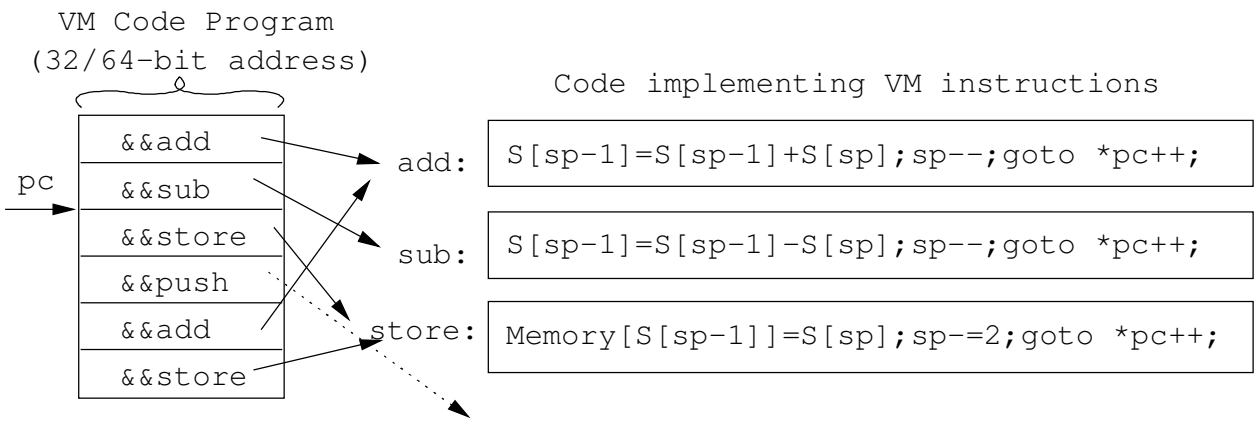
- Each instruction is represented by the address (label) of the code that implements it.
- At the end of each piece of code is an indirect jump `goto *pc++` to the next instruction.
- `"&&"` takes the address of a label. `goto *V` jumps to the label whose address is stored in variable `V`. This is a `gcc` extensions to C.

Direct Threading…

```
typedef void *Inst
static Inst prog[]={&&add,&&sub,···};

void engine() {
   Inst *pc = &prog;
   int Stack[100]; int sp=0;
   goto **pc++;

   add:  Stack[sp-1]+=Stack[sp]; sp--; goto **pc++;

   sub:  Stack[sp-1]-=Stack[sp]; sp--; goto **pc++;
}
```

Direct Threading…

- Direct threading is the most efficient method for instruction dispatch.



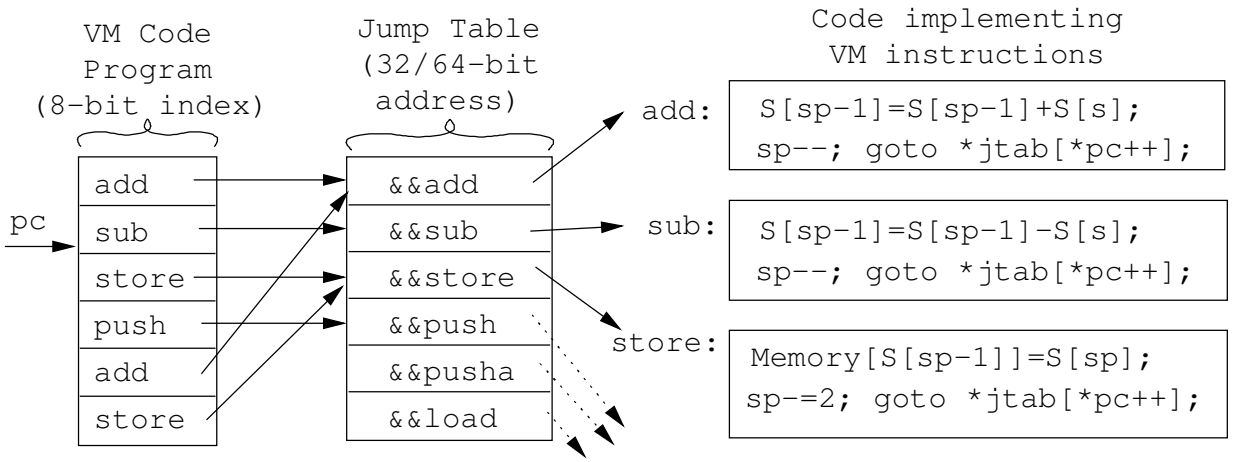
Indirect Threading

- Unfortunately, a direct threaded program will be large since each instruction is an address (32 or 64 bits).
- At the cost of an extra indirection, we can use byte-code instructions instead.
- `prog` is an array of bytes.
- `jtab` is an array of addresses of instructions.
- `goto *jtab[*pc++]` finds the current instruction (what `pc` points to), uses this to index `jtab` to get the address of the instruction, jumps to this code, and finally increments `pc`.

Indirect Threading…

```
typedef enum {add,load,···} Inst;
typedef void *Addr;
static Inst prog[]={add,sub,···};

void engine() {
   static Addr jtab[]= {&&add,&&load,···};
   Inst *pc = &prog;
   int Stack[100]; int sp=0;
   goto *jtab[*pc++];

   add:  Stack[sp-1]+=Stack[sp]; sp--;
         goto *jtab[*pc++];
}
```

Indirect Threading…



Other Optimizations

Minimizing Stack Accesses

- To reduce the cost of stack manipulation we can keep one or more of the *Top-Of-Stack* elements in registers.
- In the example below, `TOS` holds the top stack element. `Stack[sp]` holds the element second to the top, etc.

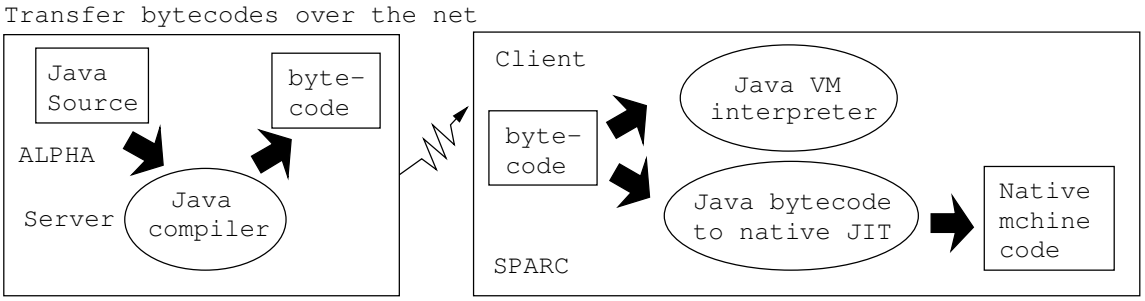
```
void engine() {
  static Inst prog[]={&&add,&&store,···};
  Inst *pc = &prog; int sp; register int TOS;
  goto *pc++;
  add:   TOS+=Stack[sp]; sp--; goto *pc++;
  store: Memory[Stack[sp]]=TOS; TOS=Stack[sp-1]
         sp-=2; goto *pc++;
}
```

Instruction Sets Revisited

- We can (sometimes) speed up the interpreter by being clever when we design the VM instruction set:
  1. Combine often used code sequences into one instruction. E.g. `muladd` a, b, c, d for $a := b * c + d$. This will reduce the number of instructions executed, but will make the VM engine larger.
  2. Reduce the total number of instructions, by making them simple and RISC-like. This will increase the number of instructions executed, but will make the VM engine smaller.
- A small VM engine may fit better in the cache than a large one, and hence yield better overall performance.

Just-In-Time Compilation

- Used to be called *Dynamic Compilation* before the marketing department got their hands on it. Also a verb, *jitting*.
- The VM code is compiled to native code just prior to execution. Gives machine independence (the bytecode can be sent over the net) and speed.
- When? When a class/module is loaded? The first time a method/procedure is called? The 2nd time it's called?



Summary

- Direct threading is the most efficient dispatch method. It cannot be implemented in ANSI C. Gnu C's "labels as values" do the trick.
- Indirect threading is almost as fast as direct threading. It may sometimes even be faster, since the interpreted program is smaller and may hence fits better in the cache.
- Call threading is the slowest method. There is overhead from the jump, save/restore of registers, the return, as well as the fact that VM registers have to be global.

Summary…

- Switch threading is slow but has the advantage to work in all languages with a case statement.
- The interpretation overhead consists of *dispatch overhead* (the cost of fetching, decoding, and starting the next instruction) and *argument access overhead*.
- You can get rid of some of the argument access overhead by *caching* the top k elements of the stack in registers. See Ertl's article.
- Jitting is difficult on machines with separate data and code caches. We must generate code into the data cache, then do a *cache flush*, then jump into the new code. Without the flush we'd be loading the old data into the code cache!

Programming Languages: Design and Implementation

Functional Programming

CE 40364

Fall 1399

Session 3

PROGRAMMING PARADIGMS

Programming Paradigms

- During the next few weeks we are going to work with functional programming. Before I can explain to you what FP is, I thought I'd better put things into perspective by talking about other programming paradigms.
- Over the last 40 or so years, a number of programming paradigms (a programming paradigm is a way to think about programs and programming) have emerged.

Programming Paradigms...

A programming paradigm

- is a way to think about programs, programming, and problem solving,
- is supported by one or more programming languages.

Being familiar with several paradigms makes you a better programmer and problem solver. The most popular paradigms:

1. Imperative programming.
2. Functional programming.
3. Object-oriented programming.
4. Logic Programming.

When all you have is a hammer, everything looks like a nail.

Programming Paradigms...

Imperative Programming

- Programming with state.
- Also known as procedural programming. The first to emerge in the 1940s-50s. Still the way most people learn how to program.
- FORTRAN, Pascal, C, BASIC.

Functional Programming

- Programming with values.
- Arrived in the late 50s with the LISP language. LISP is still popular and widely used by AI people.
- LISP, Miranda, Haskell, Gofer.

Programming Paradigms...

Object-Oriented Programming

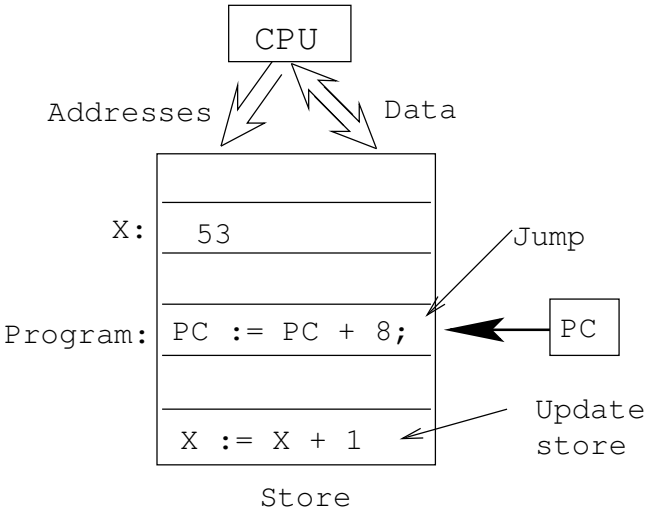
- Programming with objects that encapsulate data and operations.
- A variant of imperative programming first introduced with the Norwegian language Simula in the mid 60s.
- Simula, Eiffel, Modula-3, C++.

Logic Programming

- Programming with relations.
- Introduced in the early 70s. Based on predicate calculus. Prolog is popular with Computational Linguists.
- Prolog, Parlog.

Procedural Programming

We program an abstraction of the Von Neumann Machine, consisting of a store (memory), a program (kept in the store), A CPU and a program counter (PC):



Computing `x:=x+1`

1. Compute `x`'s address, send it to the store, get `x`'s value back.
2. Add 1 to `x`'s value.
3. Send `x`'s address and new value to the store for storage.
4. Increment `PC`.

Procedural Programming...

The programmer...

- uses control structures (IF, WHILE, ...) to alter the program counter (PC),
- uses assignment statements to alter the store.
- is in charge of memory management, i.e. declaring variables to hold values during the computation.

```
function fact (n:integer):integer;
var s,i : integer := 1;
begin
   while i<=n do s:=s*i; i:=i+1; end;
   return s;
end fact.
```

Procedural Programming…

Procedural programming is difficult because:

1. A procedural program can be in a large number of states. (Any combination of variable values and PC locations constitutes a possible state.) The programmer has to keep track of all of them.
2. Any global variable can be changed from any location in the program. (This is particularly true of languages like C & C++ [Why?]).
3. It is difficult to reason mathematically about a procedural program.

Functional Programming

In contrast to procedural languages, functional programs don't concern themselves with state and memory locations. Instead, they work exclusively with values, and expressions and functions which compute values.

- Functional programming is not tied to the von Neumann machine.
- It is not necessary to know anything about the underlying hardware when writing a functional program, the way you do when writing an imperative program.
- Functional programs are more declarative than procedural ones; i.e. they describe what is to be computed rather than how it should be computed.

Functional Languages

Common characteristics of functional programming languages:

1. Simple and concise syntax and semantics.
2. Repetition is expressed as recursion rather than iteration.
3. Functions are first class objects. I.e. functions can be manipulated just as easily as integers, floats, etc. in other languages.
4. Data as functions. I.e. we can build a function on the fly and then execute it. (Some languages).

Functional Languages…

5. Higher-order functions. I.e. functions can take functions as arguments and return functions as results.
6. Lazy evaluation. Expressions are evaluated only when needed. This allows us to build infinite data structures, where only the parts we need are actually constructed. (Some languages).
7. Garbage Collection. Dynamic memory that is no longer needed is automatically reclaimed by the system. GC is also available in some imperative languages (Modula-3, Eiffel) but not in others (C, C++, Pascal).

Functional Languages…

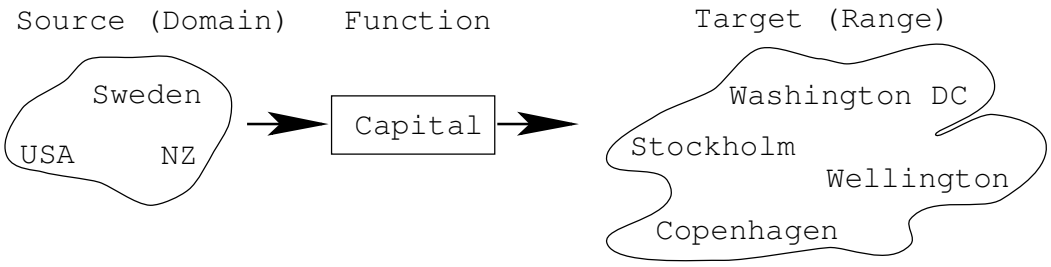
8. Polymorphic types. Functions can work on data of different types. (Some languages).
9. Functional programs can be more easily manipulated mathematically than procedural programs.

Pure vs. Impure FPL

- Some functional languages are pure, i.e. they contain no imperative features at all. Examples: Haskell, Miranda, Gofer.
- Impure languages may have assignment-statements, goto:s, while-loops, etc. Examples: LISP, ML, Scheme.

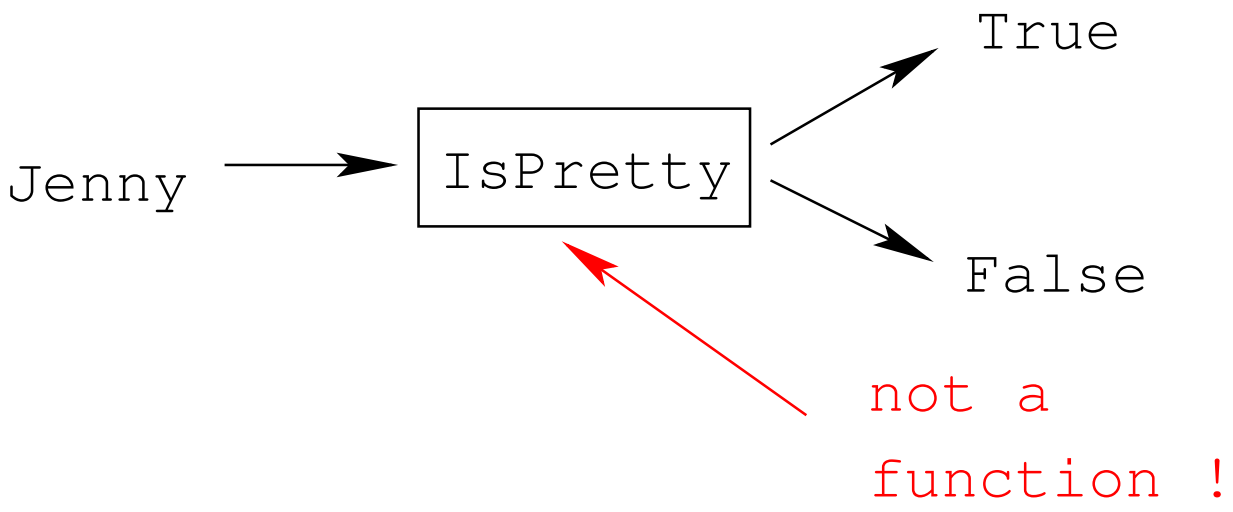
What is a function?

- A function maps argument values (inputs) to result values (outputs).
- A function takes argument values from a source set (or domain).
- A function produces result values that lie in a target set (or range).



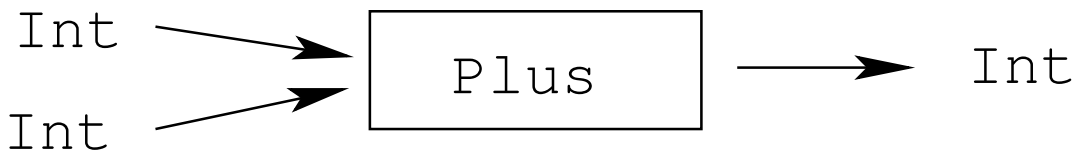
More on functions

- A function must not map an input value to more than one output value. Example:



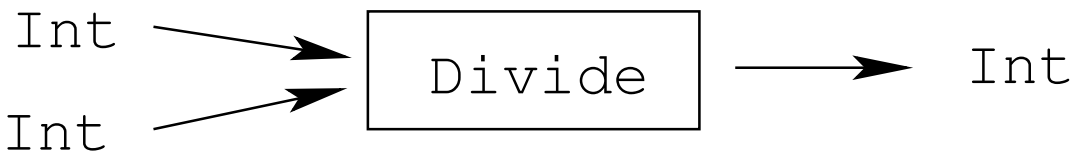
More on functions...

- If a function F maps every element in the domain to some element in the range, then F is total. I.e. a total function is defined for all arguments.



More on functions...

- A function that is undefined for some inputs, is called partial.



- `Divide` is partial since $\frac{?}{0} = ?$ is undefined.

Specifying functions

A function can be specified extensionally or intentionally.

Extensionally:

- Enumerate the elements of the (often infinite) set of pairs "`(argument, result)`" or "`Argument` $\mapsto$ `Result`."
- The extensional view emphasizes the external behavior (or specification), i.e. what the function does, rather than how it does it.

```
double = {···, (1,2), (5,10), ···}
even = {···, (0,True), (1,False), ···}
double = {···, 1↦2, 5↦10, ···}
isHandsome={Chris↦True,Hugh↦False}
```

Specifying functions…

Intensionally:

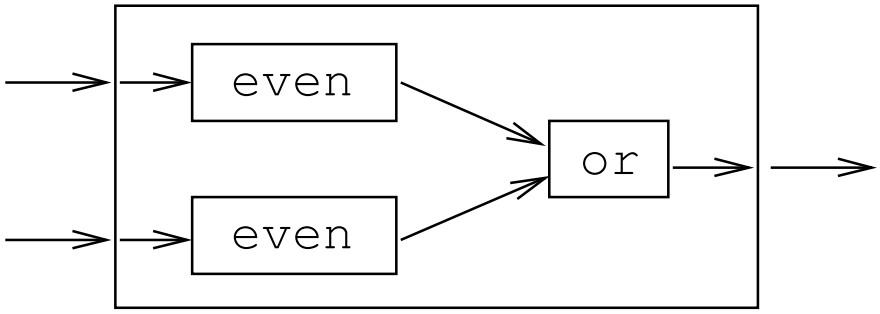
- Give a rule (i.e. algorithm) that computes the result from the arguments.
- The intentional view emphasizes the process (or algorithm) that is used to compute the result from the arguments.

```
double x = 2 * x
even x = x mod 2 == 0
isHandsome x = if isBald x
         then True
         else False
```

Specifying functions…

Graphically:

- The graphical view is a notational variant of the intentional view.



eithereven

Function Application

- The most important operation in a functional program is function application, i.e. applying an input argument to the function, and retrieving the result:

```
double x = 2 * x
even x = x mod 2 == 0

double 5 ⇒ 10
even 6 ⇒ True
```

Function Composition

- Function composition makes the result of one function application the input to another application:

```
double x = 2 * x
even x = x mod 2 == 0

even (double 5) ⇒ even 10 ⇒ True
```

Function Definition — Example

Example: How many numbers are there between m and n, inclusive?

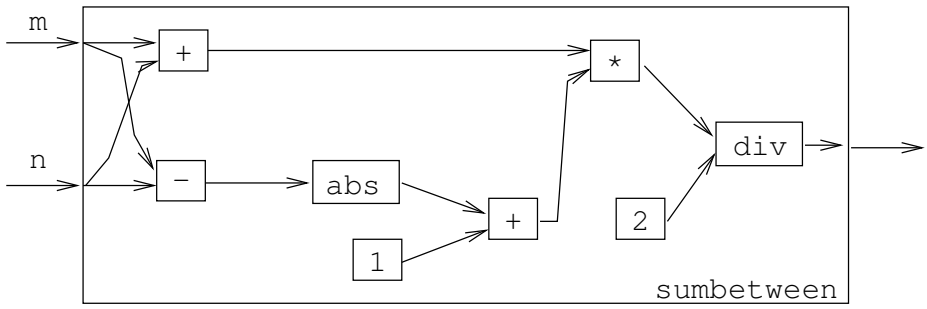
Extensional Definition:

`sumbetween m n =` $\{\cdots\ (1,1) \mapsto 1, (1,2) \mapsto 2, \cdots, (2,10) \mapsto 9\}$

Intentional Definition:

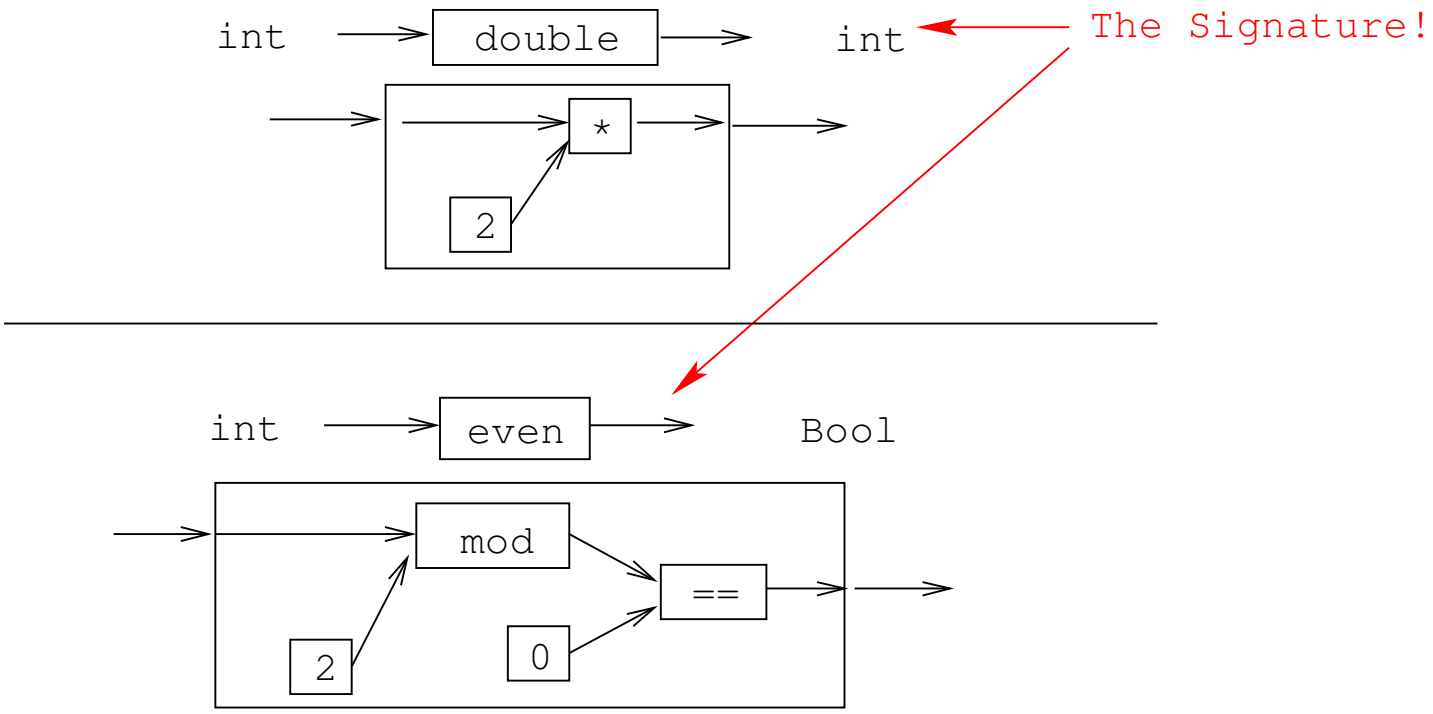
```
sumbetween m n = ((m + n) * (abs (m-n) + 1)) div 2
```

Graphical Definition:



Function Signatures

To define a function we must specify the types of the input and output sets (domain and range, i.e. the function's signature), and an algorithm that maps inputs to outputs.



Referential Transparency

- The most important concept of functional programming is referential transparency. Consider the expression

$$(2 * 3) + 5 * (2 * 3)$$

- $(2 * 3)$ occurs twice in the expression, but it has the same meaning (6) both times.
- RT means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs.
- This concept occurs naturally in mathematics, but is broken by imperative programming languages.
- RT makes functional programs easier to reason about mathematically.

Referential Transparency…

- RT is a fancy term for a very simple concept.
- What makes FP particularly simple, direct, and expressive, is that there is only one mechanism (function application) for communicating between subprograms.
- In imperative programs we can communicate either through procedure arguments or updates of global variables. This is hard to reason about mathematically.
- A notation (programming language) where the value of an expression depends only on the values of the sub-expressions is called referentially transparent.

Referential Transparency…

- Pure functional programming languages are referentially transparent.
- This means that it is easy to find the meaning (value) of an expression.
- We can evaluate it by substitution. I.e. we can replace a function application by the function definition itself.

Referential Transparency…

Evaluate `even (double 5)`:

```
double x = 2 * x
even x = x mod 2 == 0
```

```
even (double 5) ⇒
      even (2 * 5) ⇒
      even 10 ⇒
      10 mod 2 == 0 ⇒
      0 == 0 ⇒ True
```

Referential Transparency…

So, isn't Pascal referentially transparent??? Well, sometimes, yes, but not always. If a Pascal function f has side-effects (updating a global variable, doing input or output), then $f(3) + f(3)$ may not be the same as $2 * f(3)$. I.e. The second $f(3)$ has a different meaning than the first one.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;
```

```
begin
   G := 0;
   print f(3)+f(3);
   {prints 4+5=9}
end.
```

```
begin
   G := 0;
   print 2*f(3);
   {prints 2*4=8}
end.
```

Referential Transparency…

Furthermore, in many imperative languages the order in which the arguments to a binary operator are evaluated are undefined.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;
```

```
Left f(3) evaluated first.
begin
   G := 0;
   print f(3)-f(4);
   prints 4-6=-2
end.
```

```
Right f(3) evaluated first.
begin
   G := 0;
   print f(3)-f(4);
   prints 5-5=0
end.
```

Referential Transparency…

This cannot happen in a pure functional language.

1. Expressions and sub-expressions always have the same value, regardless of the environment in which they're evaluated.
2. The order in which sub-expressions are evaluated doesn't effect the final result.
3. Functions have no side-effects.
4. There are no global variables.

Referential Transparency…

5. Variables are similar to variables in mathematics: they hold a value, but they can't be updated.
6. Variables aren't (updatable) containers the way they are imperative languages.
7. Hence, functional languages are much more like mathematics than imperative languages. Functional programs can be treated as mathematical text, and manipulated using common algebraic laws.

Homework

- Here is a mathematical definition of the combinatorial function $\binom{n}{r}$ "n choose r", which computes the number of ways to pick r objects from n:

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \frac{n!}{r! * (n-r)!}$$

- Give an extensional, intentional, and graphical definition of the combinatorial function, using the notations suggested in this lecture.
- You may want to start by defining an auxiliary function to compute the factorial function, $n! = 1 * 2 * \cdots * n$.

Scheme & Racket - Introduction

Background

- Scheme is based on LISP which was developed by John McCarthy in the mid 50s.
- LISP stands for *LISt Processing*, not *Lots of Irritating Silly Parentheses*.
- Functions and data share the same representation: S-Expressions.
- A basic LISP implementation needs six functions `cons, car, cdr, equal, atom, cond`.
- Scheme was developed by Sussman and Steele in 1975.

S-Expressions

- An S-Expression is a balanced list of parentheses.

More formally, an S-expression is

1. a literal (i.e., number, boolean, symbol, character, string, or empty list).
2. a list of s-expressions.

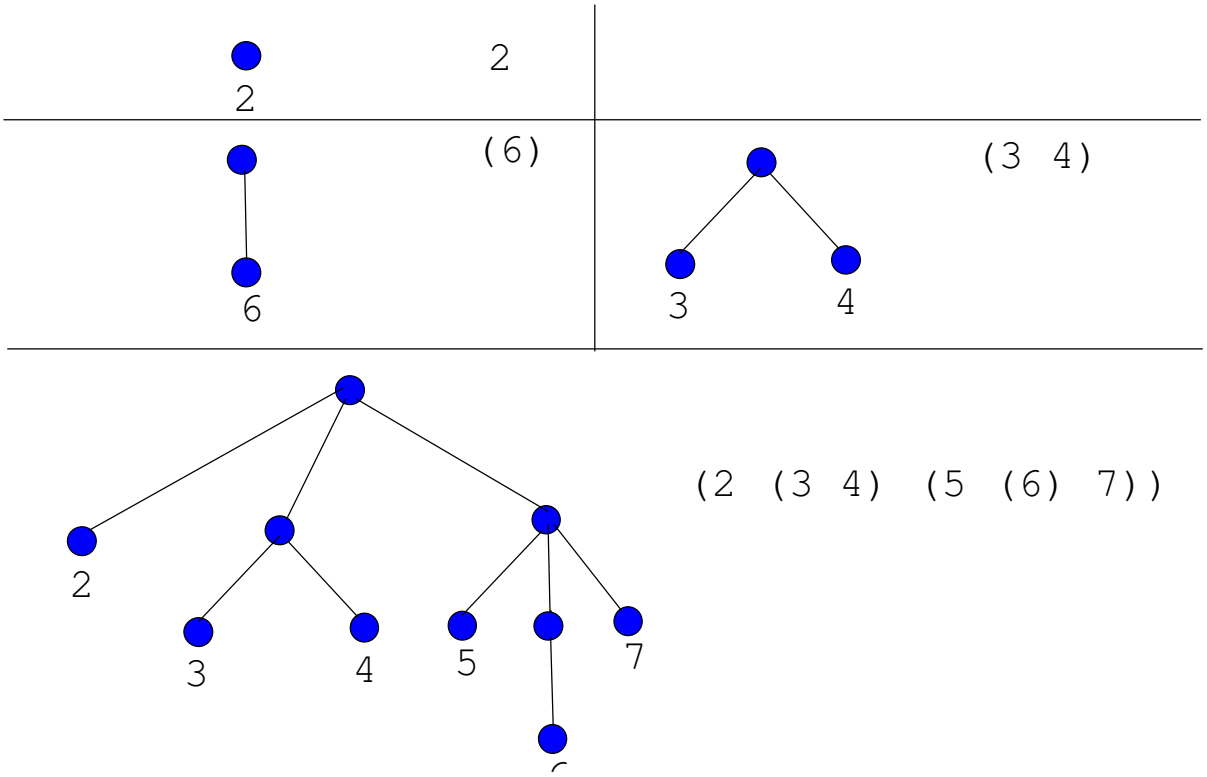
- Literals are sometimes called atoms.

S-Expressions — Examples

Legal	Illegal
`66`	`(`
`()`	`(5))`
`(4 5)`	`()()`
`((5))`	`(4 (5)`
`(()())`	`)(`
`((4 5) (6 (7)))`	

S-Expressions as Trees

- An S-expression can be seen as a linear representation of tree-structure:



S-Expressions as Function Calls

- A special case of an S-expression is when the first element of a list is a function name.
- Such an expression can be evaluated.

```
> (+ 4 5)
9
> (add-five-to-my-argument 20)
25
> (draw-a-circle 20 45)
#t
```

S-Expressions as Functions

- As we will see, function definitions are also S-expressions:

```
(define (farenheit-2-celsius f)
  (* (- f 32) 5/9)
)
```

- So, Scheme really only has one syntactic structure, the S-expression, and that is used as a data-structure (to represent lists, trees, etc), as function definitions, and as function calls.

Function Application

- In general, a function application is written like this:

$$(\texttt{operator}\ \texttt{arg}_1\ \texttt{arg}_2\ \ldots\ \texttt{arg}_n)$$

- The evaluation proceeds as follows:
  1. Evaluate `operator`. The result should be a function $\mathcal{F}$.
  2. Evaluate

     $$\texttt{arg}_1,\ \texttt{arg}_2,\ \ldots\ \texttt{arg}_n$$

     to get

     $$\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$$

  3. Apply $\mathcal{F}$ to $\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$.

Function Application — Examples

```
> (+ 4 5)
9
> (+ (+ 5 6) 3)
14
> 7
7
> (4 5 6)
eval:  4 is not a function
> #t
#t
```

Atoms — Numbers

Scheme has

- Fractions (`5/9`)
- Integers (`5435`)
- Complex numbers (`5+2i`)
- Inexact reals (`#i3.14159265`)

```
> (+ 5 4)
9
> (+ (* 5 4) 3)
23
> (+ 5/9 4/6)
1.2
> 5/9
0.5
```

Atoms — Numbers...

```
> (+ 5/9 8/18)
1
> 5+2i
5+2i
> (+ 5+2i 3-i)
8+1i
> (* 236542164521634 3746573426573425643)
886222587860913289285513763860662
> pi
#i3.141592653589793
> e
#i2.718281828459045
> (* 2 pi)
#i6.283185307179586
```

Atoms — Numbers...

- Scheme tries to do arithmetic exactly, as much as possible.
- Any computations that depend on an inexact value becomes inexact.
- Scheme has many builtin mathematical functions:

```
> (sqrt 16)
4
> (sqrt 2)
#i1.4142135623730951
> (sin 45)
#i0.8509035245341184
> (sin (/ pi 2))
#i1.0
```

Atoms — Strings

- A string is enclosed in double quotes.

```
> (display "hello")
hello
> "hello"
"hello"
> (string-length "hello")
5
> (string-append "hello" " " "world!")
"hello world!"
```

Atoms — Booleans

- `true` is written `#t`.
- `false` is written `#f`.

```
> #t
true
> #f
false
> (display #t)
#t
> (not #t)
false
```

Identifiers

- Unlike languages like C and Java, Scheme allows identifiers to contain special characters, such as `! $ % & * + - . / : < = > ? @ ^ _ ~`. Identifiers should not begin with a character that can begin a number.
- This is a consequence of Scheme's simple syntax.
- You couldn't do this in Java because then there would be many ways to interpret the expression `X-5+Y`.

Legal	Illegal
`h-e-l-l-o`	`3some`
`give-me!`	`-stance`
`WTF?`	

Defining Variables

- define binds an expression to a global name:

```
(define name expression)
```

```
(define PI 3.14)

> PI
3.14

(define High-School-PI (/ 22 7))

> High-School-PI
3.142857
```

Defining Functions

- define binds an expression to a global name:

  `(define (name arg`$_1$ `arg`$_2$ `...)  expression)`

- `arg`$_1$ `arg`$_2$ `...` are formal function parameters.

```
(define (f) 'hello)

> (f)
hello

(define (square x) (* x x))

> (square 3)
9
```

Defining Helper Functions

- A Scheme program consists of a large number of functions.
- A function typically is defined by calling other functions, so called helper or auxiliary functions.

```
(define (square x) (* x x))

(define (cube x) (* x (square x)))

> (cube 3)
27
```

Preventing Evaluation

- Sometimes you don't want an expression to be evaluated.
- For example, you may want to think of (+ 4 5) as a list of three elements `+`, `4`, and `5`, rather than as the computed value `9`.
- (quote (+ 4 5)) prevents `(+ 4 5)` from being evaluated. You can also write '(+ 4 5).

```
> (display (+ 4 5))
9
> (display (quote (+ 4 5)))
(+ 4 5)
> (display '(+ 4 5))
(+ 4 5)
```

Scheme so Far

- A function is defined by

```
(define (name arguments) expression)
```

- A variable is defined by

```
(define name expression)
```

- Strings are inclosed in double quotes, like `"this"`. Common operations on strings are
  - (string-length `string`)
  - (string-append `list-of-strings`)
- Numbers can be exact integers, inexact reals, fractions, and complex. Integers can get arbitrarily large.
- Booleans are written `#t` and `#f`.

Scheme so Far…

- An inexact number is written: `#i3.14159265`.
- Common operations on numbers are
  - (+ `arg1 arg2`), (- `arg1 arg2`)
  - (add1 `arg`), (sub1 `arg`)
  - (min `arg1 arg2`), (max `arg1 arg2`)
- A function application is written:

```
> (function-name arguments)
```

- Quoting is used to prevent evaluation

```
(quote argument)
```

or

```
'argument
```

CONDITIONAL EXPRESSIONS

Comparison Functions

- Boolean functions (by convention) end with a ?.
- We can discriminate between different kinds of numbers:

```
> (complex?  3+4i)
  #t
> (complex?  3)
  #t
> (real?  3)
  #t
> (real?  -2.5+0.0i)
  #t
> (rational?  6/10)
```

Comparison Functions…

```
  #t
> (rational?  6/3)
  #t
> (integer?  3+0i)
  #t
> (integer?  3.0)
  #t
> (integer?  8/4)
  #t
```

Tests on Numbers

- Several of the comparison functions can take multiple arguments.
- `(< 4 5 6 7 9 234)` returns true since the numbers are monotonically increasing.

```
> (< 4 5)
true
> (< 4 5 6 7 9 234)
true
> (> 5 2 1 3)
false
> (= 1 1 1 1 1)
true
> (<= 1 2 2 2 3)
true
```

Tests on Numbers...

```
> (>= 5 5)
true
> (zero?  5)
false
> (positive?  5)
true
> (negative?  5)
false
> (odd?  5)
true
> (even?  5)
false
```

Conditionals — If

- If the `test-expression` evaluates to `#f` (False) return the valuen of the `then-expression`, otherwise return the value of the `else-expression`:

```
(if test-expression
   then-expression
   else-expression
)
```

- Up to language level “Advanced Student” if-expressions must have two parts.
- Set the language level to Standard (R5RS) to get the standard Scheme behavior, where the else-expression is optional.

Conditionals — If…

```
> (define x 5)
> (if (= x 5) 2 4)
2
> (if (< x 3)
          (display "hello")
          (display "bye"))
bye
> (display
          (if (< x 3) "hello" "bye"))
bye
```

If it's not False (`#f`), it's True (`#t`)

- Any value that is not false, is interpreted as true.
- NOTE: In DrScheme this depends on which language level you set. Up to "Advanced Student", the `test-expression` of an `if` must be either `#t` or `#f`.
- Set the language level to Standard (R5RS) to get the standard Scheme behavior:

```
> (if 5 "hello" "bye")
"hello"
> (if #f "hello" "bye")
"bye"
> (if #f "hello")
> (if #t "hello")
"hello"
```

Boolean Operators

- `and` and `or` can take multiple arguments.
- `and` returns true if none of its arguments evaluate to False.
- `or` returns true if any of its arguments evaluates to True.

```
> (and (< 3 5) (odd? 5) (inexact? (cos 32)))
#t
> (or (even? 5) (zero? (- 5 5)))
#t
> (not 5)
#f
> (not #t)
#f
```

Boolean Operators...

- In general, any value that is not `#f` is considered true.
- `and` and `or` evaluate their arguments from left to right, and stop as soon as they know the final result.
- The last value evaluated is the one returned.

```
> (and "hello")
"hello"
> (and "hello" "world")
"world"
> (or "hello" "world")
"hello"
```

Defining Boolean Functions

- We can define our own boolean functions:

```
(define (big-number?  n)
      (> n 10000000)
)

> (big-number?  5)
#f
> (big-number?  3847832748327 48327)
#t >
```

Conditionals — `cond`

- `cond` is a generalization of `if`:

```
(cond
    (cond-expression1 result-expression1)
    (cond-expression2 result-expression2)
    ...
    (else else-expression))
```

- Each $\texttt{cond-expression}_i$ is evaluated in turn, until one evaluates to not False.

```
> (cond
        ((< 2 3) 4)
        ((= 2 3) 5)
        (else 6))
4
```

Conditionals — `cond`...

- To make this a bit more readable, we use square brackets around the `cond`-clauses:

```
(cond
   [cond-expr1 result-expr1]
   [cond-expr2 result-expr2]
   ...
   [else else-expression])
```

```
> (cond [#f 5] [#t 6])
6
> (cond
   [(= 4 5) "hello"]
   [(> 4 5) "goodbye"]
   [(< 4 5) "see ya!"])
"see ya!"
```

Conditionals — `case`

- `case` is like Java/C's `switch` statment:

```
(case key
    [(expr1 expr2 ...)  result-expr1]
    [(expr11 expr11 ...)  result-expr2]
    ...
    (else else-expr))
```

- The *key* is evaluated once, and compared against each *cond-expr* in turn, and the corresponding *result-expr* is returned.

```
> (case 5 [(2 3) "hello"] [(4 5) "bye"])
"bye"
```

Conditionals — `case`...

```
(define (classify n)
   (case n
      [(2 4 8 16 32) "small power of 2"]
      [(2 3 5 7 11) "small prime number"]
      [else "some other number"]
    )
 )
> (classify 4)
"small power of 2"
> (classify 3)
"small prime number"
> (classify 2)
"small power of 2"
> (classify 32476)
"some other number"
```

Sequencing

- To do more than one thing in sequence, use `begin`:

  (begin arg_1 arg_2 ...)

```
> (begin
   (display "the meaning of life=")
   (display (* 6 7))
   (newline)
)
the meaning of life=42
```

Examples — $!n$

- Write the factorial function $!n$:

```
(define (!  n)
   (cond
      [(zero?  n) 1]
      [else (* n (!  (- n 1)))]
   )
)

> (!  5)
120
```

Examples — $\binom{n}{r}$

- Write the $\binom{n}{r}$ function in Scheme:

$$\binom{n}{r} = \frac{n!}{r! * (n-r)!}$$

- Use the factorial function from the last slide.

```
(define (choose n r)
   (/ (!  n) (* (!  r) (!  (- n r))))
)

> (choose 5 2)
10
```

Examples — `(sum m n)`

- Write a function `(sum m n)` that returns the sum of the integers between `m` and `n`, inclusive.

```
(define (sum m n)
   (cond
      [(= m n) m]
      [else (+ m (sum (+ 1 m) n))]
   )
)

> (sum 1 2)
3
> (sum 1 4)
10
```

Examples — Ackermann's function

- Implement Ackermann's function:

$$
\begin{aligned}
A(1,j) &= 2j \text{ for } j \geq 1 \\
A(i,1) &= A(i-1,2) \text{ for } i \geq 2 \\
A(i,j) &= A(i-1, A(i,j-1)) \text{ for } i,j \geq 2
\end{aligned}
$$

```
(define (A i j)
   (cond
      [(and (= i 1) (>= j 1)) (* 2 j)]
      [(and (>= i 2) (= j 1)) (A (- i 1) 2)]
      [(and (>= i 2) (>= j 2))
            (A (- i 1) (A i (- j 1)))]
   )
)
```

Examples — Ackermann's function…

- Ackermann's function grows very quickly:

```
> (A 1 1)
2
> (A 3 2)
512
> (A 3 3)
1561585988519419914804999641169225
4958731641184786755447122887443528
0601470939536037485963338068553800
6371637297210170750776562389313989
2867298012168192
```

Scheme so Far

- Unlike languages like Java and C which are statically typed (we describe in the program text what type each variable is) Scheme is dynamically typed. We can test at runtime what particular type of number an atom is:
  - `(complex? arg), (real? arg)`
  - `(rational? arg), (integer? arg)`
- Tests on numbers:
  - `(< arg1, arg2), (> arg1, arg2)`
  - `(= arg1, arg2), (<= arg1, arg2)`
  - `(>= arg1, arg2), (zero? arg)`
  - `(positive? arg), (negative? arg)`
  - `(odd? arg), (even? arg)`

Scheme so Far…

- Unlike many other languages like Java which are statement-oriented, Scheme is expression-oriented. That is, every construct (even `if`, `cond`, etc) return a value. The `if-expression` returns the value of the `then-expr` or the `else-expr`:

  ```
  (if test-expr then-expr else-expr)
  ```

  depending on the value of the `test-expr`.

Scheme so Far…

- The `cond`-expression evaluates its guards until one evaluates to non-false. The corresponding value is returned:

```
(cond
    (guard1 value1)
    (guard2 value2)
    ...
    (else else-expr))
```

Scheme so Far…

- The `case`-expression evaluates `key`, finds the first matching expression, and returns the corresponding result:

```
(case key
   [(expr1 expr2 ...)  result-expr1]
   [(expr11 expr11 ...)  result-expr2]
   ...
   (else else-expr))
```

Scheme so Far…

- `and` and `or` take multiple arguments, evaluate their results left-to-right until the outcome can be determined (for `or` when the first non-`false`, for `and` when the first `false` is found), and returns the last value evaluated.

SYMBOLS AND STRUCTURES

Symbols

- In addition to numbers, strings, and booleans, Scheme has a primitive data-type (*atom*) called symbol.
- A symbol is a lot like a string. It is written:

```
'identifier
```

- Here are some examples:

```
'apple
'pear
'automobile
```

- `(symbol? arg)` checks if an atom is a symbol.
- To compare two symbols for equality, use `(eq? arg1 arg2)`. `HTDP` says to use `(symbol=? arg1 arg2)` but DrScheme doesn't seem to support this.

Symbols…

```
> (symbol?  "hello")
#f
> (symbol?  'apple)
#t
> (eq?  'a 'a)
#t
> (eq?  'a 'b)
#f
> (display 'apple)
apple
> (string->symbol "apple")
apple
> (symbol->string 'apple)
"apple"
```

Symbols…

```
(define (healthy?  f)
   (case f
      [(sushi sashimi) 'hell-yeah]
      [(coke) 'I-wish]
      [(licorice) 'no-but-yummy]
      [else 'nope]
   ))
> (healthy?  'sashimi)
hell-yeah
> (healthy?  'coke)
i-wish
> (healthy?  'licorice)
no-but-yummy
> (healthy?  'pepsi)
nope
```

Structures

- Some versions of Scheme have structures. Select Advanced Student in DrScheme.
- These are similar to C's `struct`, and Java's `class` (but without inheritance and methods).
- Use `define-struct` to define a structure:

```
(define-struct struct-name (f1 f2 ...))
```

- `define-struct` will automatically define a constructor:

```
(make-struct-name (f1 f2 ...))
```

and field-selectors:

```
struct-name-f1
struct-name-f2
```

Structures...

```
(define-struct person (name sex date-of-birth))

> (define bob (make-person "bob" 'male '1978))
> bob
(make-person "bob" 'male '1978)
> (define alice (
        make-person "alice" 'female '1979))

> (person-sex bob)
'male
> (person-date-of-birth alice)
'1979
```

Equivalence

- Every language definition has to struggle with equivalence; i.e. what does it mean for two language elements to be the same?
- In Java, consider the following example:

```
void M(String s1,String s2,int i1,int i2) {
    if (i1 == i2) ...;
    if (s1 == s2) ...;
    if (s1.equals(s2)) ...;
}
```

Why can I use `==` to compare `int`s, but it is it usually wrong to use it to compare strings?

Equivalence...

- Scheme has three equivalence predicates `eq?`, `eqv?` and `equal?`.
- `eq?` is the pickiest of the three, then comes `eqv?`, and last `equal?`.
- In other words,
  - If `(equal? a b)` returns `#t`, then so will `(eq? a b)` and `(eqv? a b)`.
  - If `(eqv? a b)` returns `#t`, then so will `(eq? a b)`..
- `(equal? a b)` generally returns `#t` if `a` and `b` are structurally the same, i.e. print the same.

Equivalence...

`(eqv? a b)` returns #t if:

- `a` and `b` are both `#t` or both `#f`.
- `a` and `b` are both symbols with the same name.
- `a` and `b` are both the same number.
- `a` and `b` are strings that denote the same locations in the store.

```
> (define S "hello")
> (eqv? S S)
true
> (eqv? "hello" "hello")
false
> (eqv? 'hello 'hello)
true
```

Equivalence…

- `(equal? a b)` returns #t if `a` and `b` are strings that print the same.
- This is known as structural equivalence.

```
> (equal? "hello" "hello")
true
> (equal? alice bob)
false
> (define alice1 (
        make-person "alice" 'female '1979))
> (define alice2 (
        make-person "alice" 'female '1979))
> (equal? alice1 alice2)
true
```

LIST PROCESSING

Constructing Lists

- The most important data structure in Scheme is the list.
- Lists are constructed using the function `cons`:

  `(cons first rest)`

  `cons` returns a list where the first element is `first`, followed by the elements from the list `rest`.

```
> (cons 'a '())
(a)
> (cons 'a (cons 'b '()))
(a b)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
```

Constructing Lists...

- There are a variety of short-hands for constructing lists.
- Lists are heterogeneous, they can contain elements of different types, including other lists.

```
> '(a b c)
(a b c)
> (list 'a 'b 'c)
(a b c)

> '(1 a "hello")
(1 a "hello")
```

Examining Lists

- `(car L)` returns the first element of a list. Some implementations also define this as `(first L)`.
- `(cdr L)` returns the list `L`, without the first element. Some implementations also define this as `(rest L)`.
- Note that `car` and `cdr` do not destroy the list, just return its parts.

```
> (car '(a b c))
'a
> (cdr '(a b c))
'(b c)
```

Examining Lists...

- Note that `(cdr L)` always returns a list.

```
> (car (cdr '(a b c)))
'b
> (cdr '(a b c))
'(b c)
> (cdr (cdr '(a b c)))
'(c)
> (cdr (cdr (cdr '(a b c))))
'()
> (cdr (cdr (cdr (cdr '(a b c)))))
error
```

Examining Lists...

- A shorthand has been developed for looking deep into a list:

```
(clist of "a" and "d"r L)
```

  Each `"a"` stands for a `car`, each `"d"` for a `cdr`.

- For example, `(caddar L)` stands for

```
(car (cdr (cdr (car L))))
```

```
> (cadr '(a b c))
'b
> (cddr '(a b c))
'(c)
> (caddr '(a b c))
'c
```

Lists of Lists

- Any S-expression is a valid list in Scheme.
- That is, lists can contain lists, which can contain lists, which...

```
> '(a (b c))
(a (b c))
> '(1 "hello" ("bye" 1/4 (apple)))
(1 "hello" ("bye" 1/4 (apple)))
> (caaddr '(1 "hello" ("bye" 1/4 (apple))))
"bye"
```

List Equivalence

- `(equal? L1 L2)` does a structural comparison of two lists, returning `#t` if they “look the same”.
- `(eqv? L1 L2)` does a “pointer comparison”, returning `#t` if two lists are “the same object”.

```
> (eqv? '(a b c) '(a b c))
false
> (equal? '(a b c) '(a b c))
true
```

List Equivalence…

- This is sometimes referred to as deep equivalence vs. shallow equivalence.

```
> (define myList '(a b c))
> (eqv?  myList myList)
true
> (eqv?  '(a (b c (d))) '(a (b c (d))))
false
> (equal?  '(a (b c (d))) '(a (b c (d))))
true
```

Predicates on Lists

- `(null? L)` returns `#t` for an empty list.
- `(list? L)` returns `#t` if the argument is a list.

```
> (null? '())
#t
> (null? '(a b c))
#f
> (list? '(a b c))
#t
> (list? "(a b c)")
#f
```

List Functions — Examples...

```
> (memq 'z '(x y z w))
#t
> (car (cdr (car '((a) b (c d)))))
(c d)
> (caddr '((a) b (c d)))
(c d)
> (cons 'a '())
(a)
> (cons 'd '(e))
(d e)
> (cons '(a b) '(c d))
((a b) (c d))
```

Recursion over Lists — cdr-recursion

- Compute the length of a list.
- This is called cdr-recursion.

```
(define (length x)
   (cond
      [(null?  x) 0]
      [else (+ 1 (length (cdr x)))]
   )
)

> (length '(1 2 3))
3
> (length '(a (b c) (d e f)))
3
```

Recursion over Lists — car-cdr-recursion

- Count the number of atoms in an S-expression.
- This is called car-cdr-recursion.

```
(define (atomcount x)
   (cond
      [(null?  x) 0]
      [(list?  x)
            (+ (atomcount (car x))
               (atomcount (cdr x)))]
      [else 1]
   ))
> (atomcount '(1))
1
> (atomcount '("hello" a b (c 1 (d))))
6
```

Recursion Over Lists — Returning a List

- Map a list of numbers to a new list of their absolute values.
- In the previous examples we returned an atom — here we're mapping a list to a new list.

```
(define (abs-list L)
   (cond
      [(null?  L) '()]
      [else (cons (abs (car L))
                  (abs-list (cdr L)))]
   )
)

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```

Recursion Over Two Lists

- `(atom-list-eq? L1 L2)` returns `#t` if `L1` and `L2` are the same list of atoms.

```
(define (atom-list-eq? L1 L2)
   (cond
      [(and (null? L1) (null? L2)) #t]
      [(or (null? L1) (null? L2)) #f]
      [else (and
         (atom? (car L1))
         (atom? (car L2))
         (eqv? (car L1) (car L2))
         (atom-list-eq? (cdr L1) (cdr L2)))]
   )
)
```

Recursion Over Two Lists…

```
> (atom-list-eq?  '(1 2 3) '(1 2 3))
#t
> (atom-list-eq?  '(1 2 3) '(1 2 a))
#f
```

Append

```
(define (append L1 L2)
   (cond
      [(null?  L1) L2]
      [else
         (cons (car L1)
            (append (cdr L1) L2))]
   )
)

> (append '(1 2) '(3 4))
(1 2 3 4)
> (append '() '(3 4))
(3 4)
> (append '(1 2) '())
(1 2)
```

Deep Recursion — `equal?`

```
(define (equal? x y)
   (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x))
         (not (atom? y))
         (equal? (car x) (car y))
         (equal? (cdr x) (cdr y)))))
```

```
> (equal? 'a 'a)
#t
> (equal? '(a) '(a))
#t
> (equal? '((a)) '((a)))
#t
```

Patterns of Recursion — cdr-recursion

- We process the elements of the list one at a time.
- Nested lists are not descended into.

```
(define (fun L)
   (cond
      [(null?  L) return-value]
      [else ...(car L) ...(fun (cdr L)) ...]
   )
)
```

Patterns of Recursion — car-cdr-recursion

- We descend into nested lists, processing every atom.

```
(define (fun x)
   (cond
      [(null?  x) return-value]
      [(atom?  x) return-value]
      [(list?  x)
            ...(fun (car x)) ...
            ...(fun (cdr x)) ...]
      [else return-value]
   ))
```

Patterns of Recursion — Maps

- Here we map one list to another.

```
(define (map L)
   (cond
      [(null?  L) '()]
      [else (cons (...(car L) ...)
                  (map (cdr L)))]
   )
)
```

Example: Binary Trees

- A binary tree can be represented as nested lists:

  `(4 (2 () () ( 6 ( 5 () ()) ()))))`

- Each node is represented by a triple

  (*data left-subtree right-subtree*)

- Empty subtrees are represented by `()`.

Example: Binary Trees...

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (caddr tree))

(define (print-spaces N)
   (cond
      [(= N 0) ""]
      [else (begin
         (display " ")
         (print-spaces (- N 1))))))

(define (print-tree tree)
   (print-tree-rec tree 0))
```

Example: Binary Trees...

```
(define (print-tree-rec tree D)
   (cond
      [(null?  tree)]
      [else (begin
         (print-spaces D)
         (display (key tree)) (newline)
         (print-tree-rec (left tree) (+ D 1))
         (print-tree-rec (right tree) (+ D 1))
)]))

> (print-tree '(4 (2 () ()) (6 (5 () ()) ())))
4
   2
   6
      5
```

Binary Trees using Structures

- We can use structures to define tree nodes.

```
(define-struct node (data left right))

(define (tree-member x T)
   (cond
      [(null?  T) #f]
      [(= x (node-data T)) #t]
      [(< x (node-data T))
         (tree-member x (node-left T))]
      [else
         (tree-member x (node-right T))]
   )
)
```

Binary Trees using Structures...

```
(define tree
    (make-node 4
       (make-node 2 '() '())
       (make-node 6
          (make-node 5 '() '())
          (make-node 9 '() '()))))

> (tree-member 4 tree)
true
> (tree-member 5 tree)
true
> (tree-member 19 tree)
false
```

Homework

- Write a function `swapFirstTwo` which swaps the first two elements of a list. Example: `(1 2 3 4)` $\Rightarrow$ `(2 1 3 4)`.
- Write a function `swapTwoInLists` which, given a list of lists, forms a new list of all elements in all lists, with first two of each swapped. Example: `((1 2 3) (4) (5 6))` $\Rightarrow$ `(2 1 3 4 6 5)`.

HIGHER-ORDER FUNCTIONS

Higher-Order Functions

- A function is higher-order if
  1. it takes another function as an argument, or
  2. it returns a function as its result.
- Functional programs make extensive use of higher-order functions to make programs smaller and more elegant.
- We use higher-order functions to encapsulate common patterns of computation.

Higher-Order Functions: `map`

- Map a list of numbers to a new list of their absolute values.
- Here's the definition of `abs-list` from a previous lecture:

```
(define (abs-list L)
   (cond
      [(null?  L) '()]
      [else (cons (abs (car L))
                  (abs-list (cdr L)))]
   )
)

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```

Higher-Order Functions: `map`...

- This type of computation is very common.
- Scheme therefore has a built-in function

```
(map f L)
```

which constructs a new list by applying the function f to every element of the list L.

$$\texttt{(map f '(}e_1\ e_2\ e_3\ e_4\texttt{))}$$
$$\Downarrow$$
$$\texttt{((f }e_1\texttt{) (f }e_2\texttt{) (f }e_3\texttt{) (f }e_4\texttt{)))}$$

Higher-Order Functions: `map`...

- `map` is a higher-order function, i.e. it takes another function as an argument.

```
(define (addone a) (+ 1 a))

> (map addone '(1 2 3)
(2 3 4)

> (map abs '(-1 2 -3))
(1 2 3)
```

Higher-Order Functions: `map`...

- We can easily define `map` ourselves:

```
(define (mymap f L)
   (cond
      [(null?  L) '()]
      [else
         (cons (f (car L)) (mymap f (cdr L)))])
```

```
> (mymap abs '(-1 2 -3))
(1 2 3)
```

Higher-Order Functions: `map`...

- If the function takes n arguments, we give `map` n lists of arguments:

```
> (map string-append
          '("A" "B" "C") '("1" "2" "3"))
("A1" "B2" "C3")

> (map + '(1 2 3) '(1 2 3))
(list 2 4 6)

> (map cons '(a b c) '((1) (2) (3)))
((a 1) (b 2) (c 3))
```

Lambda Expressions

- A lambda-expression evaluates to a function:

```
(lambda (x) (* x x))
```

  `x` is the function's formal parameter.

- Lambda-expressions don't give the function a name — they're anonymous functions.
- Evaluating the function:

```
> ((lambda (x) (* x x)) 3)
9
```

Higher-Order Functions: `map`...

- We can use `lambda`-expressions to construct anonymous functions to pass to `map`. This saves us from having to define auxiliary functions:

```
(define (addone a) (+ 1 a))

> (map addone '(1 2 3)
(2 3 4)

> (map (lambda (a) (+ 1 a)) '(1 2 3))
(2 3 4)
```

Higher-Order Functions: `filter`

- The filter-function applies a predicate (boolean-valued function) `p` to all the elements of a list.
- A new list is returned consisting of those elements for which `p` returns `#t`.

```
(define (filter p L)
   (cond
      [(null?  L) '()]
      [(p (car L))
            (cons (car L) (filter p (cdr L)))]
      [else (filter p (cdr L))]))
```

```
> (filter (lambda (x) (> x 0)) '(1 -2 3 -4))
(1 3)
```

Higher-Order Functions: `fold`

Consider the following two functions:

```
(define (sum L)
   (cond
      [(null?  L) 0]
      [else (+ (car L) (sum (cdr L)))]))
(define (concat L)
   (cond
      [(null?  L) "" ]
      [else (string-append (car L) (concat (cdr L)))]))
```

```
> (sum '(1 2 3))
6
> (concat '("1" "2" "3"))
"123"
```

Higher-Order Functions: `fold`...

- The two functions only differ in what operations they apply (`+` vs. `string-append`, and in the value returned for the base case (`0` vs. `""`).
- The `fold` function abstracts this computation:

```
(define (fold L f n)
   (cond
      [(null?  L) n]
      [else (f (car L) (fold (cdr L) f n))]))

> (fold '(1 2 3) + 0)
6
> (fold '("A" "B" "C") string-append "")
"ABC"
```

Higher-Order Functions: `fold`

- In other words, `fold` folds a list together by successively applying the function `f` to the elements of the list `L`.

$$\texttt{(apply f '(}e_1\ e_2\ e_3\ e_4\texttt{))} \Rightarrow \texttt{(f}\ e_1\ \texttt{(f}\ e_2\ \texttt{(f}\ e_3\ e_4\texttt{)))}$$

METACIRCULAR INTERPRETATION

Introduction

- In this lecture I'm going to show how you can define Scheme by writing a metacircular interpreter for the language, i.e. an interpreter for Scheme written in Scheme.
- Before we can do that, we first need to learn a few more this about the language

Let Expressions

- A let-expression binds names to values:

  (let ((name$_1$ value$_1$) (name$_2$ value$_2$) ...)
  expression)

- The first argument to `let` is a list of `(name value)` pairs. The second argument is the expression to evaluate.

```
> (let ((a 3) (b 4) (square (lambda (x)(* x x))
        (plus +))
      (sqrt (plus (square a) (square b))))
5.0
```

Let Expressions...

- `Let`-expressions can be nested:

```
> (let ((x 5) (c 4))
    (let ((v (* 4 x))
          (t (* 2 c)))
            (+ v t)))
28
```

Imperative Features

- Scheme is an impure functional language.
- I.e., Scheme has imperative features.
- I.e., in Scheme it is possible to program with side-effects.

`(set! var value)` Change the value of `var` to `value`.

`(set-car! var value)` Change the `car`-field of the cons-cell `var` to `value`.

`(set-cdr! var value)` Change the `cdr`-field of the cons-cell `var` to `value`.

Imperative Features...

- Example:

```
> (let ((x 2) (l '(a b)))
    (set!  x 3)
    (set-car!  l '(c d))
    (set-cdr!  l '(e))
    (display x) (newline)
    (display l) (newline))
3
((c d) e)
```

Dotted Pairs

- S-expressions are constructed using dotted pairs.
- It is implemented as a `struct` (called a cons-cell) consisting of two fields (the size of a machine word) called `car` and `cdr`.
- We can manipulate these fields directly:

```
> '(1 . 2)
(1 . 2)
> (cons "stacy's" "mom")
("stacy's" . "mom")
> '(1 . (2 . 3))
(1 2 . 3)
> (cons 1 2)
(1 . 2)
```

Dotted Pairs…

- When the second part of a dottend pair (the `cdr`-field) is a list, and the innermost `cdr`-field is the empty list, we get a “normal” Scheme list:

```
> '(1 . ())
(1)
> '(1 . (2 . ()))
(1 2)
> '(1 . (2 3))
(1 2 3)
```

Dotted Pairs...

- We can use `set-car!` and `set-cdr!` to manipulate the fields of a `cons`-cell directly:

```
> (define x '(1 . 2))
> (set-car! x 'a)
> x
(a . 2)
> (set-cdr! x '(2 3))
> x
(a 2 3)
```

Dotted Pairs…

- `(cons A B)` can be thought of as first creating a `cons`-cell on the heap (using `malloc`, for example), and then setting the `car` and `cdr` fields to `A` and `B`, respectively:

```
> (define x (cons 0 0))
> x
(0 . 0)
> (set-car! x '1)
> (set-cdr! x '())
> x
(1)
```

Loops

Scheme's "for-loop" `do` takes these arguments:

1. A list of triples (*var init update*) which declares a variable *var*, with an initial value *init*, and which gets updated using the expression *update*, on each iteration;
2. A pair (*termination_cond return_value*) which gives the termination condition and return value of the loop; and
3. a loop body:

```
(do ((var1 init1 update1)
     (var12 init2 update2)
          ...
     )
     (termination_cond return_value)
  loop_body
)
```

Loops…

- Sum the numbers 1 to 4, printing out intermediate results:

```
> (do ((i 1 (+ i 1))
     (sum 0 (+ sum i)))
     ((= i 5) sum)
     (display sum)
     (newline)
)
0
1
3
6

10
```

Association Lists

- Association lists are simply lists of *key-value* pairs that can be searched sequentially:

```
> (assoc 'bob '((bob 22) (joe 32) (bob 3)))
(bob 22)
```

- The list is searchedy the list from beginning to end, returning the first pair with a matching key:

  `(assoc key alist)` Search for *key*; compare using `equal?`.

  `(assq key alist)` Search for *key*; compare using `eq?`.

  `(assv key alist)` Search for *key*; compare using `eqv?`.

Association Lists...

```
> (define e '((a 1) (b 2) (c 3)))
> (assq 'a e)
(a 1)
> (assq 'b e)
(b 2)
> (assq 'd e)
#f
> (assq (list 'a) '(((a)) ((b)) ((c))))
#f
> (assoc (list 'a) '(((a)) ((b)) ((c))))
((a))
> (assv 5 '((2 3) (5 7) (11 13)))
(5 7)
```

Association Lists...

- We can actually have more than one value:

```
> (assoc 'bob '((bob 5 male)
                (jane 32 'female)))
(bob 5 male)
```

Apply

- Apply returns the result of applying its first argument to its second argument.

```
> (apply + '(6 7))
13
> (apply max '(2 5 1 7))
7
```

Eval

- (eval *arg*) evaluates its argument.

```
> (eval '(+ 4 5))
9
> (eval '(cons 'a '(b c))) (a b c)
```

Eval...

- `eval` and `quote` are each other's inverses:

```
> (eval ''(+ 4 5))
(+ 4 5)
> (eval (eval ''(+ 4 5)))
9
> (eval (eval (eval '''(+ 4 5))))
9
```

Programs as Data

- Scheme is homoiconic, self-representing, i.e. programs and data are both represented the same (as S-expressions).
- This allows us to write programs that generate programs - useful in AI, for example.

```
> (define x 'car)
> (define y ''(a b c))
> (define p (list x y))
> p
(car '(a b c))
> (eval p)
a
```

Evaluation Order

- So far, we have said that to evaluate an expression `(op arg1 arg2 arg3)` we first evaluate the arguments, then apply the operator `op` to the resulting values.
- This is known as applicative-order evaluation.
- Example:

```
(define (double x) (* x x))

> (double (* 3 4))
   ⇒ (double 12)
   ⇒ (+ 12 12)
   ⇒ 24
```

Evaluation Order…

- This is not the only possible order of evaluation
- In normal-order evaluation parameters to a function are always passed unevaluated.
- This sometimes leads to extra work:

```
(define (double x) (* x x))

> (double (* 3 4))
  ⇒ (+ (* 3 4) (* 3 4)))
  ⇒ (+ 12 (* 3 4))
  ⇒ (+ 12 12)
  ⇒ 24
```

Evaluation Order…

- Applicative-order can sometimes also lead to more work than normal-order:

```
(define (switch x a b c)
   (cond
      ((< x 0) a)
      ((= x 0) b)
      ((> x 0) c)))

> (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
```

- Here, applicative-order evaluates all the arguments, although only one value will ever be needed.

Evaluation Order…

- Ordinary Scheme functions (such as `+`, `car`, etc) use applicative-order evaluation.
- Some special forms (`cond`, `if`, etc) must use normal order since they need to consume their arguments unevaluated:

```
> (if #t (display 5) (display 6))
5
> (cond (#f (display 5))
        (#f (display 6))
        (#t (display 7)))
7
```

A Metacircular Interpreter

- One way to define the semantics of a language (the effects that programs written in the language will have), is to write a metacircular interpreter.
- I.e, we define the language by writing an interpreter for it, in the language itself.
- A metacircular interpreter for Scheme consists of two mutually recursive functions, `mEval` and `mApply`:

```
(define (mEval Expr)
   ...
)
(define (mApply Op Args)
   ...
)
```

A Metacircular Interpreter…

- We want to be able to call our interpreter like this:

```
> (mEval (+ 1 2))
3
> (mEval (+ 1 (* 3 4)))
13
> (mEval (quote (2 3)))
(2 3)
> (mEval (car (quote (1 2))))
1
```

A Metacircular Interpreter…

```
> (mEval (cdr (quote (1 2))))
(2)
> (mEval (cons (quote 5) (quote (1 2))))
(5 1 2)
> (mEval (null?  (quote (1 2))))
#f
> (mEval (null?  (quote ())))
#t
> (mEval (if (eq?  1 1) 5 6))
5
```

A Metacircular Interpreter…

- `mEval` handles primitive special forms (`lambda`, `if`, `const`, `define`, `quote`, etc), itself.
- Note that, for these forms, we must use normal-order evaluation.
- For other expressions, `mEval` evaluates all arguments and calls `mApply` to perform the required operation:

A Metacircular Interpreter…

```
(define (mEval Expr)
   (cond
      [(null?  Expr) '()]
      [(number?  Expr) Expr]
      [(eq?  (car Expr) 'if)
         (mEvalIf (cadr Expr)
                  (caddr Expr)
                  (cadddr Expr))]
      [(eq?  (car Expr) 'quote) (cadr Expr)]
      [else (mApply (car Expr)
                  (mEvalList (cdr Expr)))]
   )
)
```

A Metacircular Interpreter…

- `mApply` checks if the operation is one of the builtin primitive ones, and if so performs the required operation:

```
(define (mApply Op Args)
   (case Op
      [(car) (caar Args)]
      [(cdr) (cdar Args)]
      [(cons) (cons (car Args) (cadr Args))]
      [(eq?)  (eq?  (car Args) (cadr Args))]
      [(null?)  (null?  (car Args))]
      [(+) (+ (car Args) (cadr Args))]
      [(*) (* (car Args) (cadr Args))]
   )
)
```

A Metacircular Interpreter…

- Some auxiliary functions:

```
(define (mEvalIf b t e)
   (if (mEval b) (mEval t) (mEval e))
)

(define (mEvalList List)
   (cond
      [(null?  List) '()]
      [else (cons (mEval (car List))
            (mEvalList (cdr List)))]
   )
)
```

A Metacircular Interpreter…

- Note that this little interpreter lacks many of Scheme's functions.
- We don't have symbols, `lambda`, `define`.
- We can't define or invoke user-defined functions.
- There are no way to define or lookup variables, local or global. To do that, `mEval` and `mApply` pass around environments (*association lists*) of variable/value pairs.

Inductive Sets of Data

Essentials of Programming Languages (Chapter 1)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Recursively Specified Data

- We introduce the basic programming tools to write interpreters, ...
- The syntax of a language is usually a nested or treelike structure
  - Recursion will be at the core of our techniques
- We must know
  - What kinds of values may occur as arguments to the procedure?
  - What kinds of values are legal for the procedure to return?

Often these sets of values are complex

Inductive Specification (top-down definition)

Powerful method for specifying a set of values

✓ $S \subseteq \{0, 1, 2, \dots\}$

Definition 1.1.1 *A natural number n is in S if and only if*

1. $n = 0$, *or*
2. $n - 3 \in S$.

If n is a natural number and is not a multiple of 3, then $n \notin S$.

Inductive Specification

in-S? : $N \rightarrow Bool$
usage: (in-S? n) = #t if n is in S, #f otherwise

```
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (if (>= (- n 3) 0)
        (in-S? (- n 3))
        #f))))
```

in-S? : $N \rightarrow Bool$ is a comment, called the *contract*

Inductive Specification (bottom-up definition)

Definition 1.1.2 *Define the set S to be the smallest set contained in N and satisfying the following two properties:*

1. $0 \in S$, *and*
2. *if* $n \in S$, *then* $n + 3 \in S$.

We need "smallest set":

- Otherwise there are many sets that satisfy the remaining two conditions

Rules of Inference

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

- Each entry is called a *rule of inference*, or just a *rule*
- The horizontal line is read as an "*if-then*"
- The part above the line is called the *hypothesis*
- The part below the line is called the *conclusion* or the *consequent*
- Hypotheses are connected by an implicit "and"
- A rule with no hypotheses is called an *axiom*

Rules of Inference

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

- A natural number n is in S if and only if the statement "$n \in S$" can be derived from the axioms by using the rules of inference finitely many times
- This makes S the smallest set that is closed under the rules
- These definitions all say the same thing.
  - the first version: a top-down definition
  - the second version: a bottom-up definition
  - the third version: a rules-of-inference version

Inductive Specification

Definition 1.1.3 (list of integers, top-down) *A Scheme list is a* list of integers *if and only if either*

1. *it is the empty list, or*
2. *it is a pair whose car is an integer and whose cdr is a list of integers.*

Definition 1.1.4 (list of integers, bottom-up) *The set List-of-Int is the smallest set of Scheme lists satisfying the following two properties:*

1. $() \in$ *List-of-Int, and*
2. *if* $n \in$ *Int and* $l \in$ *List-of-Int, then* $(n \ . \ l) \in$ *List-of-Int.*

Rules of Inference

Definition 1.1.5 (list of integers, rules of inference)

$$() \in \textit{List-of-Int}$$

$$\frac{n \in \textit{Int} \qquad l \in \textit{List-of-Int}}{(n\ .\ l) \in \textit{List-of-Int}}$$

1. `()` is a list of integers, because of property 1 of definition 1.1.4 or the first rule of definition 1.1.5.

2. `(14 . ())` is a list of integers, because of property 2 of definition 1.1.4, since `14` is an integer and `()` is a list of integers. We can also write this as an instance of the second rule for *List-of-Int*.

$$\frac{\texttt{14} \in \textit{Int} \qquad () \in \textit{List-of-Int}}{(\texttt{14}\ .\ ()) \in \textit{List-of-Int}}$$

Rules of Inference

3. `(3 . (14 . ()))` is a list of integers, because of property 2, since `3` is an integer and `(14 . ())` is a list of integers. We can write this as another instance of the second rule for *List-of-Int*.

$$\frac{\texttt{3} \in \mathit{Int} \qquad \texttt{(14 . ())} \in \mathit{List\text{-}of\text{-}Int}}{\texttt{(3 . (14 . ()))} \in \mathit{List\text{-}of\text{-}Int}}$$

4. `(-7 . (3 . (14 . ())))` is a list of integers, because of property 2, since `-7` is a integer and `(3 . (14 . ()))` is a list of integers. Once more we can write this as an instance of the second rule for *List-of-Int*.

$$\frac{\texttt{-7} \in \mathit{Int} \qquad \texttt{(3 . (14 . ()))} \in \mathit{List\text{-}of\text{-}Int}}{\texttt{(-7 . (3 . (14 . ())))} \in \mathit{List\text{-}of\text{-}Int}}$$

5. Nothing is a list of integers unless it is built in this fashion.

Rules of Inference

We can combine the rules to get a picture of the entire chain of reasoning, called a *derivation* or *deduction tree*

$$\dfrac{\texttt{-7} \in N \qquad \dfrac{\texttt{3} \in N \qquad \dfrac{\texttt{14} \in N \qquad \texttt{()} \in \textit{List-of-Int}}{\texttt{(14 . ())} \in \textit{List-of-Int}}}{\texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}}{\texttt{(-7 . (3 . (14 . ())))} \in \textit{List-of-Int}}$$

Defining Sets Using Grammars

$$\begin{aligned} \textit{List-of-Int} &::= \ (\ ) \\ \textit{List-of-Int} &::= \ (\textit{Int} \quad . \quad \textit{List-of-Int}) \end{aligned}$$

- In this definition, we have:
  - Nonterminal Symbols
  - Terminal Symbols
  - Productions

Defining Sets Using Grammars

$$\begin{aligned} \textit{List-of-Int} &::= \texttt{()} \\ &::= \texttt{(}\textit{Int}\ \texttt{.}\ \textit{List-of-Int}\texttt{)} \end{aligned}$$

$$\begin{aligned} \textit{List-of-Int} &::= \texttt{()} \\ \textit{List-of-Int} &::= \texttt{(}\textit{Int}\ \texttt{.}\ \textit{List-of-Int}\texttt{)} \end{aligned}$$

$$\textit{List-of-Int} ::= \texttt{()} \mid \texttt{(}\textit{Int}\ \texttt{.}\ \textit{List-of-Int}\texttt{)}$$

$$\textit{List-of-Int} ::= \texttt{(}\{\textit{Int}\}^*\texttt{)}$$

Defining Sets Using Grammars

- *Kleene star*
  - expressed by $\{\ldots\}^*$
  - a sequence of zero or more instances of whatever appears between the braces
  - a sequence with finite length

- *Kleene plus*
  - expressed by $\{\ldots\}^+$
  - a sequence of one or more instances

- *Separated list*
  - expressed by $\{\ldots\}^{*(C)}$

Defining Sets Using Grammars

For example, $\{Int\}^{*(,)}$ includes the strings

```
8
14, 12
7, 3, 14, 16
```

and $\{Int\}^{*(;)}$ includes the strings

```
8
14; 12
7; 3; 14; 16
```

Defining Sets Using Grammars

A *syntactic derivation* may be used to show that a given data value is a member of the set

- starts with the nonterminal corresponding to the set
- at each step, a nonterminal is replaced

$$
\begin{aligned}
& \textit{List-of-Int} \\
\Rightarrow\ & (\textit{Int}\ .\ \textit{List-of-Int}) \\
\Rightarrow\ & (14\ .\ \textit{List-of-Int}) \\
\Rightarrow\ & (14\ .\ ())
\end{aligned}
$$

$$
\begin{aligned}
& \textit{List-of-Int} \\
\Rightarrow\ & (\textit{Int}\ .\ \textit{List-of-Int}) \\
\Rightarrow\ & (\textit{Int}\ .\ ()) \\
\Rightarrow\ & (14\ .\ ())
\end{aligned}
$$

Other Useful Sets

Definition 1.1.6 (s-list, s-exp)

$$\textit{S-list} ::= (\{\textit{S-exp}\}^*)$$
$$\textit{S-exp} ::= \textit{Symbol} \mid \textit{S-list}$$

Definition 1.1.7 (binary tree)

$$\textit{Bintree} ::= \textit{Int} \mid (\textit{Symbol}\ \ \textit{Bintree}\ \ \textit{Bintree})$$

Here are some examples of such trees:

```
1
2
(foo 1 2)
(bar 1 (foo 1 2))
(baz
  (bar 1 (foo 1 2))
  (biz 4 5))
```

Other Useful Sets

The *lambda calculus* is a simple language that consists only of:

- variable references
- procedures that take a single argument
- procedure calls

Definition 1.1.8 (lambda expression)

$$
\begin{aligned}
\mathit{LcExp} &::= \mathit{Identifier} \\
&::= \texttt{(lambda (}\mathit{Identifier}\texttt{) } \mathit{LcExp}\texttt{)} \\
&::= \texttt{(}\mathit{LcExp}\ \mathit{LcExp}\texttt{)}
\end{aligned}
$$

where an identifier is any symbol other than `lambda`.

Lambda Calculus

The identifier in the second production is called the *bound variable*

- It binds or captures any occurrences of the variable in the body
- Any occurrence of that variable in the body refers to this one

```
(lambda (x) (+ x 5))
```

```
((lambda (x) (+ x 5)) (- x 7))
```

Context-free Grammars

- These grammars are said to be *context-free*
  - Their rules may be applied in any context
  - Sometimes this is not restrictive enough

$$\textit{Binary-search-tree} ::= \texttt{()} \mid \texttt{(}\textit{Int}\ \ \textit{Binary-search-tree}\ \ \textit{Binary-search-tree}\texttt{)}$$

- Binary search trees:
  - All the keys in the left subtree are less than (or equal to) the key in the current node
  - All the keys in the right subtree are greater than the key in the current node

- To determine whether a particular production can be applied in a particular syntactic derivation:
  - We have to look at the context
  - Such constraints are called context-sensitive constraints or invariants

Context-sensitive Constraints

- Context-sensitive constraints also arise when specifying the syntax of programming languages
  - In many languages every variable must be declared
  - This constraint on the use of variables is sensitive to the context

- Formal methods can be used to specify context-sensitive constraints, but these methods are far more complicated

- The usual approach is first to specify a context-free grammar. Context-sensitive constraints are then added using other methods

Induction

Theorem 1.1.1 *Let t be a binary tree, as defined in definition 1.1.7. Then t contains an odd number of nodes.*

The key to the proof is that the substructures of a tree t are always smaller than t itself. This pattern of proof is called *structural induction*.

Proof by Structural Induction

To prove that a proposition IH(s) is true for all structures s, prove the following:

1. *IH is true on simple structures (those without substructures).*
2. *If IH is true on the substructures of s, then it is true on s itself.*

Deriving Recursive Programs

The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

nth-element

Procedure *list-ref* takes a list, *lst* , and a zero-based index, *n*, and returns element number *n* of *lst*

```
> (list-ref '(a b c) 1)
b
```

nth-element : *List* × *Int* → *SchemeVal*
usage: (nth-element *lst* *n*) = the *n*-th element of *lst*

```
(define nth-element
  (lambda (lst n)
    (if (null? lst)
      (report-list-too-short n)
      (if (zero? n)
        (car lst)
        (nth-element (cdr lst) (- n 1))))))

(define report-list-too-short
  (lambda (n)
    (eopl:error 'nth-element
      "List too short by ~s elements.~%" (+ n 1))))
```

List ::= () | (*Scheme value* . *List*)

```
  (nth-element '(a b c d e) 3)
= (nth-element   '(b c d e) 2)
= (nth-element     '(c d e) 1)
= (nth-element       '(d e) 0)
= d
```

remove-first

The procedure *remove-first* should take two arguments: a symbol, *s*, and a list of symbols, *los*

The first occurrence of the symbol *s* should be removed

remove-first : *Sym* × *Listof(Sym)* → *Listof(Sym)*

```
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      (if (eqv? (car los) s)
        (cdr los)
        (cons (car los) (remove-first s (cdr los)))))))
```

List-of-Symbol ::= () | (*Symbol* . *List-of-Symbol*)

```
> (remove-first 'a '(a b c))
(b c)
> (remove-first 'b '(e f g))
(e f g)
> (remove-first 'a4 '(c1 a4 c1 a4))
(c1 c1 a4)
> (remove-first 'x '())
()
```

occurs-free?

A variable *occurs free* in an expression *exp* if it has some occurrence in *exp* that is not inside some lambda binding of the same variable

```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

LcExp ::= *Identifier*
::= (lambda (*Identifier*) *LcExp*)
::= (*LcExp* *LcExp*)

occurs-free?

We can summarize these cases in the rules:

- If the expression e is a variable, then the variable x occurs free in e if and only if x is the same as e.
- If the expression e is of the form `(lambda (`y`) `e'`)`, then the variable x occurs free in e if and only if y is different from x and x occurs free in e'.
- If the expression e is of the form `(`e_1 e_2`)`, then x occurs free in e if and only if it occurs free in e_1 or e_2. Here, we use "or" to mean *inclusive or*, meaning that this includes the possibility that x occurs free in both e_1 and e_2. We will generally use "or" in this sense.

occurs-free?

occurs-free? : $Sym \times LcExp \rightarrow Bool$
usage: returns #t if the symbol *var* occurs free in *exp*, otherwise returns #f.

```
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
       (and
         (not (eqv? var (car (cadr exp))))
         (occurs-free? var (caddr exp))))
      (else
        (or
          (occurs-free? var (car exp))
          (occurs-free? var (cadr exp)))))))
```

subst

The procedure *subst* should take three arguments: two symbols, *new* and *old*, and an s-list, *slist*.

subst : $Sym \times Sym \times S\text{-}list \rightarrow S\text{-}list$

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

subst-in-s-exp : $Sym \times Sym \times S\text{-}exp \rightarrow S\text{-}exp$

```
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
      (if (eqv? sexp old) new sexp)
      (subst new old sexp))))
```

$$S\text{-}list ::= (\{S\text{-}exp\}^*)$$
$$S\text{-}exp ::= Symbol \mid S\text{-}list$$

$$S\text{-}list ::= ()$$
$$::= (S\text{-}exp \ . \ S\text{-}list)$$
$$S\text{-}exp ::= Symbol \mid S\text{-}list$$

```
> (subst 'a 'b '((b c) (b () d)))
((a c) (a () d))
```

Follow the Grammar!

- Write one procedure for each nonterminal in the grammar. The procedure will be responsible for handling the data corresponding to that nonterminal, and nothing else.
- In each procedure, write one alternative for each production corresponding to that nonterminal. You may need additional case structure, but this will get you started. For each nonterminal that appears in the right-hand side, write a recursive call to the procedure for that nonterminal.

Auxiliary Procedures

Procedure *number-elements* should take any list $(v_0 \quad v_1 \quad v_2 \ \ldots)$ and return the list $((0 \ \ v_0) \ \ (1 \ \ v_1) \ \ (2 \ \ v_2) \ \ldots))$

number-elements-from : $Listof(SchemeVal) \times Int \rightarrow Listof(List(Int, SchemeVal))$

usage: (number-elements-from '$(v_0 \ v_1 \ v_2 \ \ldots) \ n$)
= $((n \ v_0) \ (n+1 \ v_1) \ (n+2 \ v_2) \ \ldots)$

```
(define number-elements-from
  (lambda (lst n)
    (if (null? lst) '()
      (cons
        (list n (car lst))
        (number-elements-from (cdr lst) (+ n 1))))))
```

context argument

number-elements : $List \rightarrow Listof(List(Int, SchemeVal))$

```
(define number-elements
  (lambda (lst)
    (number-elements-from lst 0)))
```

Vectors

- Vectors are heterogeneous structures
- It occupies less space than a list of the same length
- The length of a vector is the number of elements
  - fixed when the vector is created
- The first element in a vector is indexed by zero
- Vectors are written using the notation #(object ...)

```
#(0 (2 2 2 2) "Anna")
```

Summing All the Values in a Vector

$$\sum_{i=0}^{i=length(v)-1} v_i$$

where v is a vector of integers. We generalize it by turning the upper bound into a parameter n, so that the new task is to compute

$$\sum_{i=0}^{i=n} v_i$$

where $0 \leq n < length(v)$.

Summing All the Values in a Vector

partial-vector-sum : $Vectorof(Int) \times Int \rightarrow Int$
usage: `if` $0 \leq n < length(v)$, `then`

$$\texttt{(partial-vector-sum } v\ n\texttt{)} = \sum_{i=0}^{i=n} v_i$$

```
(define partial-vector-sum
  (lambda (v n)
    (if (zero? n)
      (vector-ref v 0)
      (+ (vector-ref v n)
         (partial-vector-sum v (- n 1))))))
```

Summing All the Values in a Vector

vector-sum : $Vectorof(Int) \rightarrow Int$

usage: (vector-sum v) = $\sum_{i=0}^{i=length(v)-1} v_i$

```
(define vector-sum
  (lambda (v)
    (let ((n (vector-length v)))
      (if (zero? n)
        0
        (partial-vector-sum v (- n 1))))))
```

Data Abstraction

Essentials of Programming Languages (Chapter 2)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Specifying Data via Interfaces

- We do not want to be concerned with representations of data types
  - manipulation files
  - arithmetic operations...
- We may also decide to change the representation of the data
- Data abstraction divides a data type into:
  - an interface: It tells us
    - what the data of the type represents
    - what the operations on the data are
    - what properties these operations may be relied on to have
  - an implementation: It provides
    - a specific representation of the data
    - code for the operations that make use of that data representation

Representation

$\lceil v \rceil$: The representation of data v

- Data type of natural numbers:
  - The interface is to consist of four procedures: zero, is-zero?, successor, and predecessor

$$(\texttt{zero}) = \lceil 0 \rceil$$

$$(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$(\texttt{successor}\ \lceil n \rceil) = \lceil n+1 \rceil \quad (n \geq 0)$$

$$(\texttt{predecessor}\ \lceil n+1 \rceil) = \lceil n \rceil \quad (n \geq 0)$$

Client Program

- The client code manipulates the new data only through the operations specified in the interface
  - This code is representation-independent
- A sample of client program, manipulating natural numbers:

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
      y
      (successor (plus (predecessor x) y)))))
```

No matter what representation is in use

Interfaces

- Most interfaces contain
  - some *constructors*
    - for building elements of the data type
  - some *observers*
    - for extract information from values of the data type

- *Zero*, *successor*, and *predecessor* are constructors and *is-zero?* is observer

Representation

Some representations for natural numbers:

1) Unary representation:
   - natural number *n* is represented by a list of n #t's
   - 0 is represented by (), 1 is represented by (#t), 2 is represented by (#t #t), ...

$$\lceil 0 \rceil = \texttt{()}$$
$$\lceil n+1 \rceil = \texttt{(\#t . } \lceil n \rceil \texttt{)}$$

In this representation, we can satisfy the specification by writing

```
(define zero (lambda () '()))
(define is-zero? (lambda (n) (null? n)))
(define successor (lambda (n) (cons #t n)))
(define predecessor (lambda (n) (cdr n)))
```

Representation

2) Scheme number representation:

- we simply use Scheme's internal representation of numbers

```
(define zero (lambda () 0))
(define is-zero? (lambda (n) (zero? n)))
(define successor (lambda (n) (+ n 1)))
(define predecessor (lambda (n) (- n 1)))
```

Representation

3) Bignum representation:

- Numbers are represented in base N, for some large integer N
- The representation becomes a list consisting of numbers between 0 and N −1
  - with least-significant bigit first
- This representation makes it easy to represent integers that are much larger than can be represented in a machine word

$$\lceil n \rceil = \begin{cases} (\,) & n = 0 \\ (r \;\; . \;\; \lceil q \rceil) & n = qN + r,\ 0 \le r < N \end{cases}$$

So if $N = 16$, then $\lceil 33 \rceil = (1 \;\; 2)$ and $\lceil 258 \rceil = (2 \;\; 0 \;\; 1)$, since

$$258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$$

Opaque vs. Transparent

- Types can be divided into:
  - Opaque types
    - The representation of a type is hidden, so it cannot be exposed by any operation (including printing)
  - Transparent types
- Scheme does not provide a standard mechanism for creating new opaque types
  - we define interfaces and rely on the writer of the client program to be discreet and use only the procedures in the interfaces

Representation Strategies for Data Types

Consider a data type of *environments*.

- An environment associates a value with each element of a finite set of variables
- An environment is a function whose domain is a finite set of variables, and whose range is the set of all Scheme values
- environment $env = \{(var_1, val_1), \ldots, (var_n, val_n)\}$

The interface to this data type has three procedures, specified as follows:

$$
\begin{array}{ll}
\texttt{(empty-env)} & = \lceil \emptyset \rceil \\
\texttt{(apply-env } \lceil f \rceil \ var\texttt{)} & = f(var) \\
\texttt{(extend-env } var \ v \ \lceil f \rceil\texttt{)} & = \lceil g \rceil, \\
 & \quad \text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}
\end{array}
$$

Environment

For example, the expression

```
> (define e
    (extend-env 'd 6
      (extend-env 'y 8
        (extend-env 'x 7
          (extend-env 'y 14
            (empty-env)))))))
```

defines an environment e such that $e(\mathtt{d}) = 6$, $e(\mathtt{x}) = 7$, $e(\mathtt{y}) = 8$, and e is undefined on any other variables.

In this example, `empty-env` and `extend-env` are the constructors, and `apply-env` is the only observer

Environment

We can obtain a representation of environments by observing that every environment can be built by starting with the empty environment and applying `extend-env` n times, for some $n \geq 0$, e.g.,

```
(extend-env var_n val_n
    ...
    (extend-env var_1 val_1
      (empty-env))...)
```

So every environment can be built by an expression in the following grammar:

$$
\begin{aligned}
\textit{Env-exp} &::= \texttt{(empty-env)} \\
&::= \texttt{(extend-env } \textit{Identifier}\ \ \textit{Scheme-value}\ \ \textit{Env-exp}\texttt{)}
\end{aligned}
$$

A data-structure representation of environments

Env = (empty-env) | (extend-env Var $SchemeVal$ Env)
Var = Sym

empty-env : $() \rightarrow Env$

```
(define empty-env
  (lambda () (list 'empty-env)))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))
```

A data-structure representation of environments

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
       (let ((saved-var (cadr env))
             (saved-val (caddr env))
             (saved-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))

(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))

(define report-invalid-env
  (lambda (env)
    (eopl:error 'apply-env "Bad environment: ~s" env)))
```

Env = (empty-env) | (extend-env Var $SchemeVal$ Env)

Procedural Representation

$Env = Var \rightarrow SchemeVal$

empty-env : $() \rightarrow Env$

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var)))))
```

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
       (let ((saved-var (cadr env))
             (saved-val (caddr env))
             (saved-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))

(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))

(define report-invalid-env
  (lambda (env)
    (eopl:error 'apply-env "Bad environment: ~s" env)))
```

Interfaces for Recursive Data Types

$$
\begin{aligned}
\textit{Lc-exp} &::= \textit{Identifier} \\
&::= (\texttt{lambda}\ \ (\textit{Identifier})\ \ \textit{Lc-exp}) \\
&::= (\textit{Lc-exp}\ \ \textit{Lc-exp})
\end{aligned}
$$

The definition of occurs-free? in section 1.2.4 is not as readable as it might be

Our interface will have constructors and two kinds of observers: predicates and extractors

An Interface for Lambda calculus expressions

The constructors are:

var-exp	$: \mathit{Var} \longrightarrow \mathit{Lc\text{-}exp}$
lambda-exp	$: \mathit{Var} \times \mathit{Lc\text{-}exp} \longrightarrow \mathit{Lc\text{-}exp}$
app-exp	$: \mathit{Lc\text{-}exp} \times \mathit{Lc\text{-}exp} \longrightarrow \mathit{Lc\text{-}exp}$

The predicates are:

var-exp?	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Bool}$
lambda-exp?	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Bool}$
app-exp?	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Bool}$

Finally, the extractors are

var-exp->var	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Var}$
lambda-exp->bound-var	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Var}$
lambda-exp->body	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Lc\text{-}exp}$
app-exp->rator	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Lc\text{-}exp}$
app-exp->rand	$: \mathit{Lc\text{-}exp} \longrightarrow \mathit{Lc\text{-}exp}$

$$
\begin{aligned}
\mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\
&::= \texttt{(lambda}\ \ \texttt{(}\mathit{Identifier}\texttt{)}\ \ \mathit{Lc\text{-}exp}\texttt{)} \\
&::= \texttt{(}\mathit{Lc\text{-}exp}\ \ \mathit{Lc\text{-}exp}\texttt{)}
\end{aligned}
$$

Occurs-free?

occurs-free? : *Sym* × *LcExp* → *Bool*

```
(define occurs-free?
  (lambda (search-var exp)
    (cond
      ((var-exp? exp) (eqv? search-var (var-exp->var exp)))
      ((lambda-exp? exp)
       (and
         (not (eqv? search-var (lambda-exp->bound-var exp)))
         (occurs-free? search-var (lambda-exp->body exp))))
      (else
        (or
          (occurs-free? search-var (app-exp->rator exp))
          (occurs-free? search-var (app-exp->rand exp))))))))
```

Lc-exp ::= *Identifier*
::= (lambda (*Identifier*) *Lc-exp*)
::= (*Lc-exp* *Lc-exp*)

Interfaces for Recursive Data Types

Designing an interface for a recursive data type

1. *Include one constructor for each kind of data in the data type.*
2. *Include one predicate for each kind of data in the data type.*
3. *Include one extractor for each piece of data passed to a constructor of the data type.*

A Tool for Defining Recursive Data Types

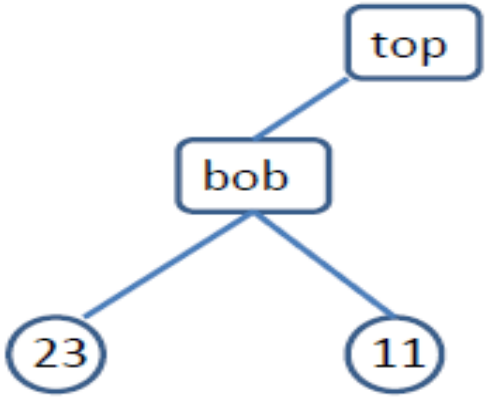
```
(require (lib "eopl.ss"  "eopl"))
```

(define-datatype *type-name* *type-predicate-name*
  { (*variant-name* { (*field-name* *predicate*) }*) }$^+$)

```
(define-datatype binaryTree binaryTree?
        (null-node)
        (leaf-node (datum number?))
        (interior-node (key symbol?)
                       (left-child binaryTree?)
                       (right-child binaryTree?)))
```

A Tool for Defining Recursive Data Types

```
(define T1 (leaf-node 23))
(define T2 (leaf-node 11))
(define T3 (interior-node 'bob T1 T2))
(define T4 (interior-node 'top T3 (null-node)))
```



Lc-exp (define-datatype)

```
(define-datatype lc-exp lc-exp?
  (var-exp
    (var identifier?))
  (lambda-exp
    (bound-var identifier?)
    (body lc-exp?))
  (app-exp
    (rator lc-exp?)
    (rand lc-exp?)))
```

$$
\begin{aligned}
\textit{Lc-exp} &::= \textit{Identifier} \\
&::= (\texttt{lambda}\ \ (\textit{Identifier})\ \ \textit{Lc-exp}) \\
&::= (\textit{Lc-exp}\ \ \textit{Lc-exp})
\end{aligned}
$$

The constructors are:

var-exp	$: \textit{Var} \longrightarrow \textit{Lc-exp}$
lambda-exp	$: \textit{Var} \times \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$
app-exp	$: \textit{Lc-exp} \times \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$

The predicates are:

var-exp?	$: \textit{Lc-exp} \longrightarrow \textit{Bool}$
lambda-exp?	$: \textit{Lc-exp} \longrightarrow \textit{Bool}$
app-exp?	$: \textit{Lc-exp} \longrightarrow \textit{Bool}$

var-exp->var	$: \textit{Lc-exp} \longrightarrow \textit{Var}$
lambda-exp->bound-var	$: \textit{Lc-exp} \longrightarrow \textit{Var}$
lambda-exp->body	$: \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$
app-exp->rator	$: \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$
app-exp->rand	$: \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$

S-list (define-datatype)

```
(define-datatype s-list s-list?
  (empty-s-list)
  (non-empty-s-list
    (first s-exp?)
    (rest s-list?)))

(define-datatype s-exp s-exp?
  (symbol-s-exp
    (sym symbol?))
  (s-list-s-exp
    (slst s-list?)))
```

$$\textit{S-list} ::= (\{\textit{S-exp}\}^*)$$
$$\textit{S-exp} ::= \textit{Symbol} \mid \textit{S-list}$$

Cases

There is also a cases expression that gives you access to the variant fields of an object constructed with define-datatype. The format of this is

```
(cases  type  object
        (variant1 (data fields) exp1)
        (variant2 (data fields) exp2)
        etc. )
```

Cases

```
(define sum (lambda (tree)
    (cases binaryTree tree
        (null-node () 0)
        (leaf-node (v) v)
        (interior-node (sym left right) (+ (sum left) (sum right))))))
```

Occurs-free (cases)

occurs-free? : $Sym \times LcExp \rightarrow Bool$

```
(define occurs-free?
  (lambda (search-var exp)
    (cases lc-exp exp
      (var-exp (var) (eqv? var search-var))
      (lambda-exp (bound-var body)
        (and
          (not (eqv? search-var bound-var))
          (occurs-free? search-var body)))
      (app-exp (rator rand)
        (or
          (occurs-free? search-var rator)
          (occurs-free? search-var rand))))))
```

Domain-specific language

- The form define-datatype is an example of a *domain-specific language*
- A domain-specific language is a small language for describing a single task
- In this case, the task was defining a recursive data type
- Such a language may lie inside a general-purpose language, as define-datatype does, or it may be a standalone language

Abstract Syntax and Its Representation

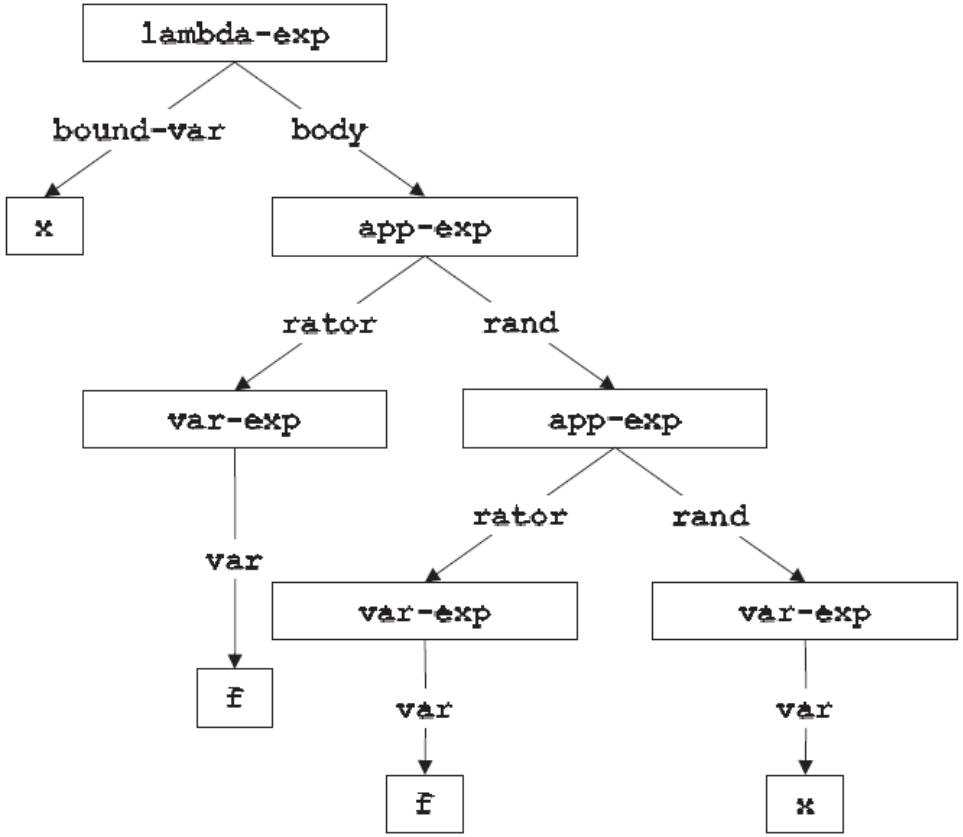
- A grammar usually specifies a particular representation of an inductive data type
  - Such a representation is called concrete syntax, or external representation

$$
\begin{aligned}
\textit{Lc-exp} &::= \textit{Identifier} \\
&::= \texttt{proc}\ \ \textit{Identifier}\ \ \texttt{=>}\ \ \textit{Lc-exp} \\
&::= \textit{Lc-exp}\ \texttt{(}\textit{Lc-exp}\texttt{)}
\end{aligned}
$$

- To process such data, we need to convert it to an internal representation
  - The define-datatype form provides a convenient way of defining such an internal representation.
  - We call this abstract syntax
  - Terminals such as parentheses need not be stored, because they convey no information

Abstract syntax tree

It is convenient to visualize the internal representation as an abstract syntax tree




```
(lambda (x) (f (f x)))
```

Abstract Syntax Tree

- Each internal node of the tree is labeled with the associated production name
- Edges are labeled with the name of the corresponding nonterminal occurrence
- Leaves correspond to terminal strings

$$
\begin{aligned}
\textit{Lc-exp} ::= \; & \textit{Identifier} \\
& \boxed{\texttt{var-exp (var)}} \\
::= \; & \texttt{(lambda (}\textit{Identifier}\texttt{)} \; \textit{Lc-exp}\texttt{)} \\
& \boxed{\texttt{lambda-exp (bound-var body)}} \\
::= \; & \texttt{(}\textit{Lc-exp} \; \textit{Lc-exp}\texttt{)} \\
& \boxed{\texttt{app-exp (rator rand)}}
\end{aligned}
$$

Create an Abstract Syntax

- To create an abstract syntax for a given concrete syntax, we must name each production of the concrete syntax and each occurrence of a nonterminal in each production
- It is straightforward to generate define-datatype declarations for the abstract syntax. We create one define-datatype for each nonterminal, with one variant for each production.

Parsing

- If the concrete syntax is a set of strings of characters, it may be a complex undertaking to derive the corresponding abstract syntax tree
- This task is called parsing and is performed by a parser
- Because writing a parser is difficult in general, it is best performed by a tool called a parser *generator*
- A parser generator takes as input a grammar and produces a parser

Converting a Concrete Syntax to Abstract Syntax

If the concrete syntax is given as a set of lists, the parsing process is considerably simplified

parse-expression : *SchemeVal* → *LcExp*

```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
         (lambda-exp
           (car (cadr datum))
           (parse-expression (caddr datum)))
         (app-exp
           (parse-expression (car datum))
           (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum)))))
```

Converting an Abstract Syntax to Concrete Syntax

It is usually straightforward to convert an abstract syntax tree back to a list-and-symbol representation. If we do this, the Scheme print routines will then display it in a list-based concrete syntax. This is performed by `unparse-lc-exp`:

unparse-lc-exp : *LcExp* → *SchemeVal*

```
(define unparse-lc-exp
  (lambda (exp)
    (cases lc-exp exp
      (var-exp (var) var)
      (lambda-exp (bound-var body)
        (list 'lambda (list bound-var)
          (unparse-lc-exp body)))
      (app-exp (rator rand)
        (list
```

Interpreter Recipe

The Interpreter Recipe

1. *Look at a piece of data.*
2. *Decide what kind of data it represents.*
3. *Extract the components of the datum and do the right thing with them.*

Expressions

Essentials of Programming Languages (Chapter 3)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Expressions

- In this chapter, we study the binding and scoping of variables
- We write specifications for our languages, and implement them using interpreters, following the interpreter recipe
- Our specifications and interpreters take a context argument, called the *environment*, which keeps track of the meaning of each variable in the expression being evaluated

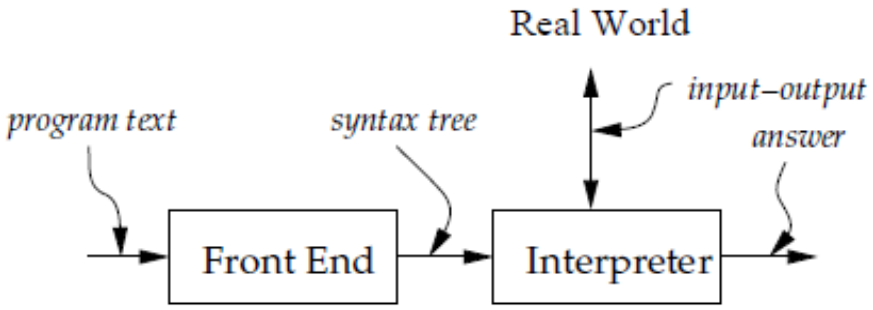
Specification and Implementation Strategy

Our specification will consist of assertions of the form

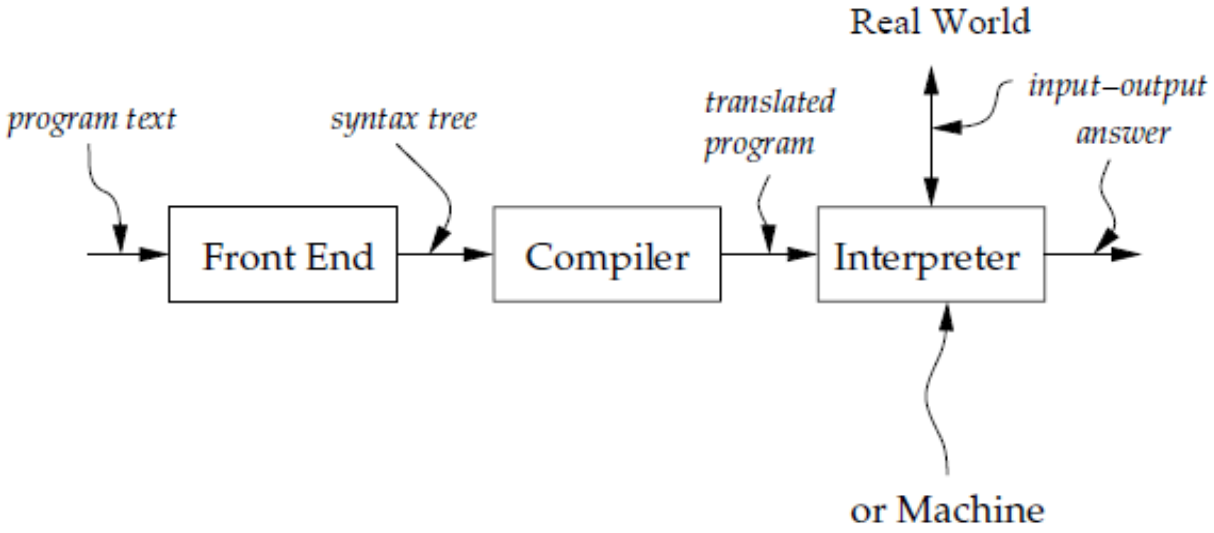
$$(\texttt{value-of}\ \ exp\ \ \rho) = val$$

meaning that the value of expression exp in environment ρ should be val. We write down rules of inference and equations, like those in chapter 1, that will enable us to derive such assertions. We use the rules and equations by hand to find the intended value of some expressions.

Figure 3.1 Block diagrams for a language-processing system



(a) Execution via interpreter



Language Processing System

- We start with the text of the program written in the language we are implementing
  - This is called the source language or the defined language
- Program text (a program in the source language) is passed through a front end
  - It converts the program to an abstract syntax tree
- The interpreter is itself written in some language
  - We call that language the implementation language or the defining language
- In figure 3.1(b), the interpreter is replaced by a compiler, which translates the abstract syntax tree into a program in some other language (the target language), and that program is executed

LET: A Simple Language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-` `(` *Expression* `,` *Expression* `)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(` *Expression* `)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

```
(scan&parse "-(55, -(x,11))")
#(struct:a-program
   #(struct:diff-exp
      #(struct:const-exp 55)
      #(struct:diff-exp
         #(struct:var-exp x)
         #(struct:const-exp 11))))
```

Specification of Values

- An important part of the specification of any programming language is the set of values that the language manipulates
- Each language has at least two such sets:
  - The expressed values
    - They are the possible values of expressions
  - The denoted values
    - They are the values bound to variables

$$ExpVal = Int + Bool$$
$$DenVal = Int + Bool$$

num-val	$: Int \rightarrow ExpVal$
bool-val	$: Bool \rightarrow ExpVal$
expval->num	$: ExpVal \rightarrow Int$
expval->bool	$: ExpVal \rightarrow Bool$

Environments

An environment is a function whose domain is a finite set of variables and whose range is the denoted values. We use some abbreviations when writing about environments.

- ρ ranges over environments.
- [] denotes the empty environment.
- $[var = val]\rho$ denotes `(extend-env` var val ρ`)`.
- $[var_1 = val_1, var_2 = val_2]\rho$ abbreviates $[var_1 = val_1]([var_2 = val_2]\rho)$, etc.
- $[var_1 = val_1, var_2 = val_2, \ldots]$ denotes the environment in which the value of var_1 is val_1, etc.

Environments

We will occasionally write down complicated environments using indentation to improve readability. For example, we might write

```
[x=3]
 [y=7]
  [u=5] ρ
```

to abbreviate

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 ρ)))
```

Specifying the Behavior of Expressions

```
(value-of (const-exp n) ρ) = (num-val n)

(value-of (var-exp var) ρ) = (apply-env ρ var)

(value-of (diff-exp exp1 exp2) ρ)
= (num-val
    (-
      (expval->num (value-of exp1 ρ))
      (expval->num (value-of exp2 ρ))))
```

constructors:

const-exp	$: Int \rightarrow Exp$
zero?-exp	$: Exp \rightarrow Exp$
if-exp	$: Exp \times Exp \times Exp \rightarrow Exp$
diff-exp	$: Exp \times Exp \rightarrow Exp$
var-exp	$: Var \rightarrow Exp$
let-exp	$: Var \times Exp \times Exp \rightarrow Exp$

observer:

value-of	$: Exp \times Env \rightarrow ExpVal$

...

We write «exp» to denote the AST for expression exp. We also write $\lceil n \rceil$ in place of `(num-val` n`)`, and $\lfloor val \rfloor$ in place of `(expval->num` val`)`. We will also use the fact that $\lfloor \lceil n \rceil \rfloor = n$.

Figure 3.3 A simple calculation using the specification

Let $\rho =$ [i=1,v=5,x=10].

```
(value-of
  <<-(-(x,3), -(v,i))>>
  ρ)

= ⌈(-
     ⌊(value-of <<-(x,3)>> ρ)⌋
     ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
     (-
       ⌊(value-of <<x>> ρ)⌋
       ⌊(value-of <<3>> ρ)⌋)
     ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
     (-
       10
       ⌊(value-of <<3>> ρ)⌋)
     (value-of <<-(v,i)>> ρ))⌉

= ⌈(-
     (-
       10
       3)
     ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
     7
     ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
     7
     (-
       ⌊(value-of <<v>> ρ)⌋
       ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
     7
     (-
       5
       ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
     7
     (-
       5
       1))⌉

= ⌈(-
     7
     4)⌉

= ⌈3⌉
```

Specifying the Behavior of Programs

In our language, a whole program is just an expression. In order to find the value of such an expression, we need to specify the values of the free variables in the program. So the value of a program is just the value of that expression in a suitable initial environment. We choose our initial environment to be `[i=1,v=5,x=10]`.

```
(value-of-program exp)
= (value-of exp [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉])
```

Specifying Conditionals

We use `bool-val` as a constructor to turn a boolean into an expressed value, and `expval->num` as an extractor to check whether an expressed value is an integer, and if so, to return the integer.

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ \ (\texttt{zero?-exp}\ \ exp_1)\ \ \rho) \\ \quad = \begin{cases}(\texttt{bool-val}\ \ \texttt{\#t}) & \text{if } (\texttt{expval->num}\ \ val_1) = 0 \\ (\texttt{bool-val}\ \ \texttt{\#f}) & \text{if } (\texttt{expval->num}\ \ val_1) \neq 0\end{cases}\end{array}}$$

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ \ (\texttt{if-exp}\ \ exp_1\ \ exp_2\ \ exp_3)\ \ \rho) \\ \quad = \begin{cases}(\texttt{value-of}\ \ exp_2\ \ \rho) & \text{if } (\texttt{expval->bool}\ \ val_1) = \texttt{\#t} \\ (\texttt{value-of}\ \ exp_3\ \ \rho) & \text{if } (\texttt{expval->bool}\ \ val_1) = \texttt{\#f}\end{cases}\end{array}}$$

...

Rules of inference like this make the intended behavior of any individual expression easy to specify, but they are not very good for displaying a deduction.

We can then use substitution of equals for equals to display a calculation.

For an `if-exp`, the equational specification is

```
(value-of (if-exp exp1 exp2 exp3) ρ)
= (if (expval->bool (value-of exp1 ρ))
      (value-of exp2 ρ)
      (value-of exp3 ρ))
```

Figure 3.4 A simple calculation for a conditional expression

Let $\rho = [x=\lceil 33 \rceil, y=\lceil 22 \rceil]$.

```
(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)

= ⌈18⌉
```

Specifying let

`let` expressions may be nested, as in

```
let z = 5
in let x = 3
   in let y = -(x,1)      % here x = 3
      in let x = 4
         in -(z, -(x,y)) % here x = 4
```

In this example, the reference to `x` in the first difference expression refers to the outer declaration, whereas the reference to `x` in the other difference expression refers to the inner declaration, and thus the entire expression's value is 3.

...

The right-hand side of the `let` is also an expression, so it can be arbitrarily complex. For example,

```
let x = 7
in let y = 2
   in let y = let x = -(x,1)
              in -(x,y)
      in -(-(x,8), y)
```

Here the x declared on the third line is bound to 6, so the value of y is 4, and the value of the entire expression is $((-1) - 4) = -5$.

...

We can write down the specification as a rule.

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho) = val_1}{\begin{array}{c}(\texttt{value-of}\ \ (\texttt{let-exp}\ \ var\ \ exp_1\ \ body)\ \ \rho) \\ =\ \ (\texttt{value-of}\ \ body\ \ [var = val_1]\rho)\end{array}}$$

As before, it is often more convenient to recast this as the equation

$$\begin{array}{l}(\texttt{value-of}\ \ (\texttt{let-exp}\ \ var\ \ exp_1\ \ body)\ \ \rho) \\ =\ \ (\texttt{value-of}\ \ body\ \ [var{=}(\texttt{value-of}\ \ exp_1\ \ \rho)]\rho)\end{array}$$

Figure 3.5 An example of `let`

```
(value-of
  <<let x = 7
    in let y = 2
       in let y = let x = -(x,1) in -(x,y)
          in -(-(x,8),y)>>
  ρ0)

= (value-of
    <<let y = 2
      in let y = let x = -(x,1) in -(x,y)
         in -(-(x,8),y)>>
    [x=⌈7⌉] ρ0)

= (value-of
    <<let y = let x = -(x,1) in -(x,y)
      in -(-(x,8),y)>>
    [y=⌈2⌉] [x=⌈7⌉] ρ0)
```

Let ρ_1 = [y=⌈2⌉] [x=⌈7⌉] ρ_0.

```
= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<let x = -(x,1) in -(x,y)>> ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=(value-of <<-(x,1)>> ρ1)]ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=⌈6⌉]ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=⌈4⌉]ρ1)

= ⌈(- (- 7 8) 4)⌉

= ⌈-5⌉
```

Figure 3.6 Syntax data types for the LET language

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?)))
```

Implementing the Specification of LET

The procedure `init-env` constructs the specified initial environment used by `value-of-program`.

init-env : $() \rightarrow Env$
usage: `(init-env) = [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉]`

```
(define init-env
  (lambda ()
    (extend-env
     'i (num-val 1)
     (extend-env
      'v (num-val 5)
      (extend-env
       'x (num-val 10)
       (empty-env))))))
```

Figure 3.7 Expressed values for the LET language

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?)))
```

expval->num : *ExpVal* → *Int*

```
(define expval->num
  (lambda (val)
    (cases expval val
      (num-val (num) num)
      (else (report-expval-extractor-error 'num val)))))
```

expval->bool : *ExpVal* → *Bool*

```
(define expval->bool
  (lambda (val)
    (cases expval val
      (bool-val (bool) bool)
      (else (report-expval-extractor-error 'bool val)))))
```

Figure 3.8 Interpreter for the LET language

run : *String* → *ExpVal*

```
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))
```

value-of-program : *Program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

value-of : *Exp* × *Env* → *ExpVal*

```
(define value-of
  (lambda (exp env)
    (cases expression exp

      (value-of (const-exp n) ρ) = n
      (const-exp (num) (num-val num))

      (value-of (var-exp var) ρ) = (apply-env ρ var)
      (var-exp (var) (apply-env env var))
```

$$\boxed{\begin{array}{l}\texttt{(value-of (diff-exp } exp_1\ exp_2\texttt{) } \rho\texttt{)} = \\ \quad \lceil\texttt{(- } \lfloor\texttt{(value-of } exp_1\ \rho\texttt{)}\rfloor\ \lfloor\texttt{(value-of } exp_2\ \rho\texttt{)}\rfloor\texttt{)}\rceil\end{array}}$$

```
(diff-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((num1 (expval->num val1))
          (num2 (expval->num val2)))
      (num-val
        (- num1 num2)))))
```

$$\frac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (zero?-exp } exp_1\texttt{) } \rho\texttt{)} \\ \quad = \begin{cases}\texttt{(bool-val \#t)} & \text{if } \texttt{(expval->num } val_1\texttt{)} = 0 \\ \texttt{(bool-val \#f)} & \text{if } \texttt{(expval->num } val_1\texttt{)} \neq 0\end{cases}\end{array}}$$

```
(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (if (zero? num1)
        (bool-val #t)
        (bool-val #f)))))
```

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ \ (\texttt{if-exp}\ \ exp_1\ \ exp_2\ \ exp_3)\ \ \rho) \\ \quad = \begin{cases} (\texttt{value-of}\ \ exp_2\ \ \rho) & \text{if } (\texttt{expval->bool}\ \ val_1) = \texttt{\#t} \\ (\texttt{value-of}\ \ exp_3\ \ \rho) & \text{if } (\texttt{expval->bool}\ \ val_1) = \texttt{\#f} \end{cases}\end{array}}$$

```
(if-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env)))
    (if (expval->bool val1)
      (value-of exp2 env)
      (value-of exp3 env))))
```

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ \ (\texttt{let-exp}\ \ var\ \ exp_1\ \ body)\ \ \rho) \\ \quad = (\texttt{value-of}\ \ body\ \ [var = val_1]\rho)\end{array}}$$

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))))
```

PROC: A Language with Procedures

We need syntax for procedure creation and calling. This is given by the productions:

$$\begin{aligned} \textit{Expression} ::= & \texttt{proc}\ \ (\textit{Identifier})\ \ \textit{Expression} \\ & \boxed{\texttt{proc-exp (var body)}} \\ \textit{Expression} ::= & (\textit{Expression}\ \ \textit{Expression}) \\ & \boxed{\texttt{call-exp (rator rand)}} \end{aligned}$$

In (`proc-exp` *var* *body*), the variable *var* is the *bound variable* or *formal parameter*. In a procedure call (`call-exp` exp_1 exp_2), the expression exp_1 is the *operator* and exp_2 is the *operand* or *actual parameter*. We use the word *argument* to refer to the value of an actual parameter.

PROC: Example

```
let f = proc (x) -(x,11)
in (f (f 77))

(proc (f) (f (f 77))
 proc (x) -(x,11))
```

The first program creates a procedure that subtracts 11 from its argument. It calls the resulting procedure `f`, and then applies `f` twice to 77, yielding the answer 55. The second program creates a procedure that takes its argument and applies it twice to 77. The program then applies this procedure to the subtract-11 procedure. The result is again 55.

PROC

We will follow the design of Scheme, and let procedures be expressed values in our language, so that

$$ExpVal = Int + Bool + Proc$$

where *Proc* is a set of values representing procedures. We will think of *Proc* as an abstract data type.

PROC

$$\begin{aligned} \textit{Expression} ::= & \texttt{proc}\ \ (\textit{Identifier})\ \ \textit{Expression} \\ & \boxed{\texttt{proc-exp (var body)}} \\ \textit{Expression} ::= & (\textit{Expression}\ \ \textit{Expression}) \\ & \boxed{\texttt{call-exp (rator rand)}} \end{aligned}$$

The lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call. Variables occurring free in the procedure should also obey the lexical binding rule.

PROC: Example

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

Here we evaluate the expression `proc (z) -(z,x)` twice. The first time we do it, `x` is bound to 200, so by the lexical scope rule, the result is a procedure that subtracts 200 from its argument. We name this procedure `f`. The second time we do it, `x` is bound to 100, so the resulting procedure should subtract 100 from its argument. We name this procedure `g`.

PROC: Specification

The specification for a `proc` expression is

```
(value-of (proc-exp var body) ρ)
= (proc-val (procedure var body ρ))
```

where `proc-val` is a constructor, like `bool-val` or `num-val`, that builds an expressed value from a *Proc*.

At a procedure call, we want to find the value of the operator and the operand. If the value of the operator is a `proc-val`, then we want to apply it to the value of the operand.

```
(value-of (call-exp rator rand) ρ)
= (let ((proc (expval->proc (value-of rator ρ)))
        (arg (value-of rand ρ)))
    (apply-procedure proc arg))
```

PROC: Specification

Last, we consider what happens when `apply-procedure` is invoked. As we have seen, the lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call. Furthermore any other variables must have the same values they had at procedure-creation time. Therefore these procedures should satisfy the condition

```
(apply-procedure (procedure var body ρ) val)
= (value-of body [var=val]ρ)
```

```
(value-of
  <<let x = 200
    in let f = proc (z) -(z,x)
       in let x = 100
          in let g = proc (z) -(z,x)
             in -((f 1), (g 1))>>
  ρ)

= (value-of
    <<let f = proc (z) -(z,x)
      in let x = 100
         in let g = proc (z) -(z,x)
            in -((f 1), (g 1))>>
    [x=⌈200⌉]ρ)


= (value-of
    <<let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))>>
    [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
     [x=⌈200⌉]ρ)

= (value-of
    <<let g = proc (z) -(z,x)
      in -((f 1), (g 1))>>
    [x=⌈100⌉]
     [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
      [x=⌈200⌉]ρ)
```

```
= (value-of
    <<-((f 1), (g 1))>>
    [g=(proc-val (procedure z <<-(z,x)>>
                   [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ))]
     [x=⌈100⌉]
      [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
       [x=⌈200⌉]ρ)

= ⌈(-
    (value-of <<(f 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ)
    (value-of <<(g 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ))⌉

= ⌈(-
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈200⌉]ρ)
      ⌈1⌉)
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ)
      ⌈1⌉))⌉
```

```
= ⌈(-
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈200⌉]ρ)
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈100⌉][f=...][x=⌈200⌉]ρ))⌉

= ⌈(- -199 -99)⌉

= ⌈-100⌉
```

PROC: Procedural Representation

According to the recipe described in section 2.2.3, we can employ a procedural representation for procedures by their action under `apply-procedure`. To do this we define `procedure` to have a value that is an implementation-language procedure that expects an argument, and returns the value required by the specification

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))
```

PROC: Procedural Representation

Therefore the entire implementation is

procedure : $Var \times Exp \times Env \rightarrow Proc$

```
(define procedure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))
```

apply-procedure : $Proc \times ExpVal \rightarrow ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (proc1 val)))
```

proc? : $SchemeVal \rightarrow Bool$

```
(define proc?
  (lambda (val)
    (procedure? val)))
```

PROC: define-datatype

Alternatively, we could use a data structure representation like that of section 2.2.2.

proc? : *SchemeVal* $\rightarrow$ *Bool*
procedure : *Var* $\times$ *Exp* $\times$ *Env* $\rightarrow$ *Proc*

```
(define-datatype proc proc?
  (procedure
    (var identifier?)
    (body expression?)
    (saved-env environment?)))
```

apply-procedure : *Proc* $\times$ *ExpVal* $\rightarrow$ *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

...

- In either implementation, we add an alternative to the data type `expval`

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
  (proc-val
    (proc proc?)))
```

and we need to add two new clauses to `value-of`

```
(proc-exp (var body)
  (proc-val (procedure var body env)))

(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc arg)))
```

LETREC: A Language with Recursive Procedures

For example:

```
letrec double(x)
        = if zero?(x) then 0 else -((double -(x,1)), -2)
in (double 6)
```

The left-hand side of a recursive declaration is the name of the recursive procedure and its bound variable. To the right of the = is the procedure body. The production for this is

Expression ::= `letrec` *Identifier* (*Identifier*) = *Expression* `in` *Expression*

```
letrec-exp (p-name b-var p-body letrec-body)
```

LETREC: Specification

The value of a `letrec` expression is the value of the body in an environment that has the desired behavior:

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

LETREC: Specification

We specify the behavior of this environment as follows: Let ρ_1 be the environment produced by `(extend-env-rec` *proc-name bound-var proc-body* ρ`)`. Then what should `(apply-env` ρ_1 *var*`)` return?

1. If the variable *var* is the same as *proc-name*, then `(apply-env` ρ_1 *var*`)` should produce a closure whose bound variable is *bound-var*, whose body is *proc-body*, and with an environment in which *proc-name* is bound to this procedure. But we already have such an environment, namely ρ_1 itself! So

   `(apply-env` ρ_1 *proc-name*`)`
   `= (proc-val (procedure` *bound-var proc-body* ρ_1`))`

2. If *var* is not the same as *proc-name*, then

   `(apply-env` ρ_1 *var*`) = (apply-env` ρ *var*`)`

```
(value-of <<letrec double(x) = if zero?(x)
                               then 0
                               else -((double -(x,1)), -2)
           in (double 6)>> ρ0)

= (value-of <<(double 6)>>
    (extend-env-rec double x <<if zero?(x) ...>> ρ0))

= (apply-procedure
    (value-of <<double>> (extend-env-rec double x
                            <<if zero?(x) ...>> ρ0))
    (value-of <<6>> (extend-env-rec double x
                      <<if zero?(x) ...>> ρ0)))

= (apply-procedure
    (procedure x <<if zero?(x) ...>>
      (extend-env-rec double x <<if zero?(x) ...>> ρ0))
    ⌈6⌉)

= (value-of
    <<if zero?(x) ...>>
   [x=⌈6⌉](extend-env-rec
                double x <<if zero?(x) ...>> ρ0))
```

Figure 3.10 A calculation with extend-env-rec

```
= (-
    (value-of
      <<(double -(x,1))>>
      [x=⌈6⌉](extend-env-rec
                      double x <<if zero?(x) ...>> ρ0))
    -2)

= (-
    (apply-procedure
      (value-of
        <<double>>
        [x=⌈6⌉](extend-env-rec
                        double x <<if zero?(x) ...>> ρ0))
      (value-of
        <<-(x,1)>>
        [x=⌈6⌉](extend-env-rec
                        double x <<if zero?(x) ...>> ρ0)))
    -2)

= (-
    (apply-procedure
      (procedure x <<if zero?(x) ...>>
        (extend-env-rec double x <<if zero?(x) ...>> ρ0))
      ⌈5⌉)
    -2)

= ...
```

Figure 3.10 A calculation with extend-env-rec

Figure 3.12 extend-env-rec added to environments.

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))

(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name b-var p-body saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var))))))
```

Scoping

In most programming languages, variables may appear in two different ways: as *references* or as *declarations*. A variable reference is a use of the variable. For example, in the Scheme expression

```
(f x y)
```

all the variables, f, x, and y, appear as references. However, in

```
(lambda (x) (+ x 3))
```

or

```
(let ((x (+ y 7))) (+ x 3))
```

the first occurrence of x is a declaration.

Scoping

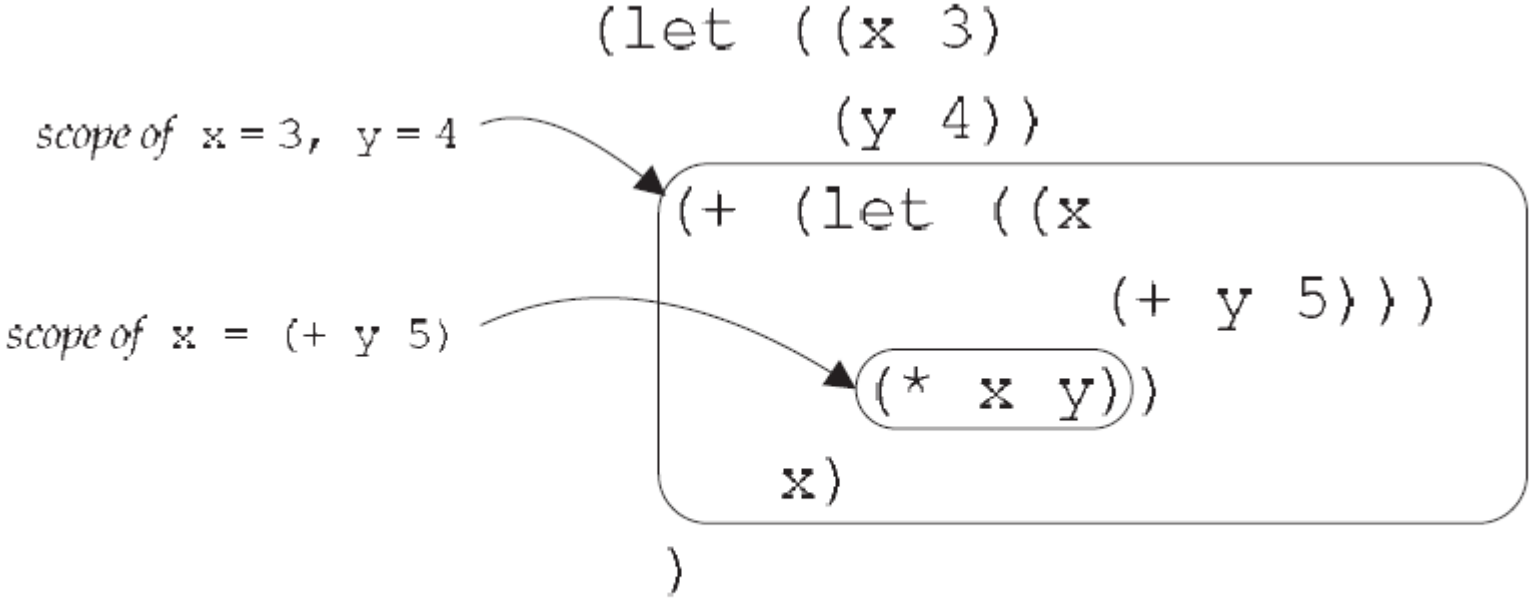


Figure 3.13 A simple contour diagram

...

To find which declaration corresponds to a given use of a variable, we search *outward* from the use until we find a declaration of the variable. Here is a simple example in Scheme.

```
(let ((x 3)                    Call this x1
      (y 4))
  (+ (let ((x                  Call this x2
             (+ y 5)))
       (* x y))                Here x refers to x2
     x))                       Here x refers to x1
```

In this example, the inner x is bound to 9, so the value of the expression is

```
(let ((x 3)
      (y 4))
  (+ (let ((x
            (+ y 5)))
       (* x y))
     x))

= (+ (let ((x
            (+ 4 5)))
       (* x 4))
    3)

= (+ (let ((x 9))
       (* x 4))
    3)

= (+ 36
    3)

= 39
```

Static properties

Every programming language must have some rules to determine the declaration to which each variable reference refers. These rules are typically called *scoping* rules. The portion of the program in which a declaration is valid is called the *scope* of the declaration.

We can determine which declaration is associated with each variable use without executing the program. Properties like this, which can be computed without executing the program, are called *static* properties.

Contour diagram: Example

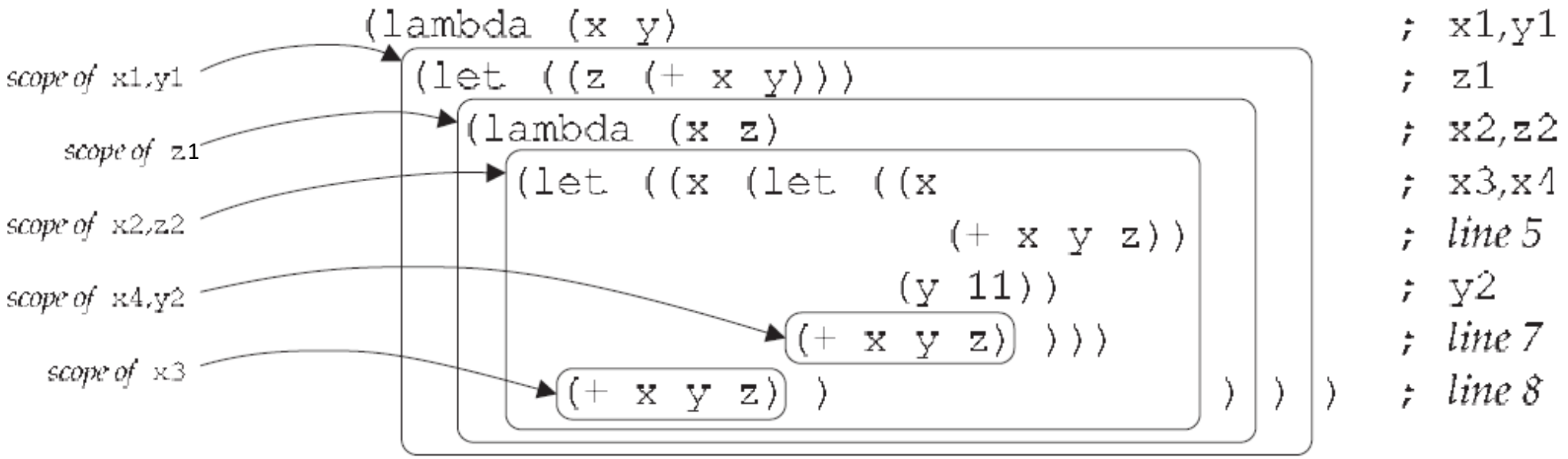


Figure 3.14 A more complicated contour diagram

...

Figure 3.14 shows a more complicated program with the contours drawn in. Here there are three occurrences of the expression `(+ x y z)`, on lines 5, 7, and 8. Line 5 is within the scope of `x2` and `z2`, which is within the scope of `z1`, which is within the scope of `x1` and `y1`. So at line 5, `x` refers to `x2`, `y` refers to `y1`, and `z` refers to `z2`. Line 7 is within the scope of `x4` and `y2`, which is within the scope of `x2` and `z2`, which is within the scope of `z1`, which is within the scope of `x1` and `y1`. So at line 7, `x` refers to `x4`, `y` refers to `y2`, and `z` refers to `z2`. Last, line 8 is within the scope of `x3`, which is within the scope of `x2` and `z2`, which is within the scope of `z1`, which is within the scope of `x1` and `y1`. So at line 8, `x` refers to `x3`, `y` refers to `y1`, and `z` refers to `z2`.

Binding

The association between a variable and its value is called a *binding*. For our language, we can look at the specification to see how the binding is created.

A variable declared by a `proc` is bound when the procedure is applied.

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))
```

A `let`-variable is bound by the value of its right-hand side.

```
(value-of (let-exp var val body) ρ)
= (value-of body (extend-env var val ρ))
```

A variable declared by a `letrec` is bound using its right-hand side as well.

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

Extent of a binding

The *extent* of a binding is the time interval during which the binding is maintained. In our little language, as in Scheme, all bindings have *semi-infinite* extent, meaning that once a variable gets bound, that binding must be maintained indefinitely (at least potentially). This is because the binding might be hidden inside a closure that is returned. In languages with semi-infinite extent, the garbage collector collects bindings when they are no longer reachable. This is only determinable at run-time, so we say that this is a *dynamic* property.

Eliminating Variable Names

The number of contours crossed is called the *lexical* (or *static*) *depth* of the variable reference. It is customary to use "zero-based indexing," thereby not counting the last contour crossed.

For example, in the Scheme expression

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

the reference to x on the last line and the reference to a have lexical depth zero, while the reference to x in the third line has lexical depth one.

Eliminating Variable Names

We could, therefore, get rid of variable names entirely, and write something like

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

```
(nameless-lambda
  ((nameless-lambda
     (#1 #0))
   #0))
```

Lexical Addresses

Here each `nameless-lambda` declares a new anonymous variable, and each variable reference is replaced by its lexical depth; this number uniquely identifies the declaration to which it refers. These numbers are called *lexical addresses* or *de Bruijn indices*. Compilers routinely calculate the lexical address of each variable reference. Once this has been done, the variable names may be discarded unless they are required to provide debugging information.

Lexical Addresses

Consider the expression

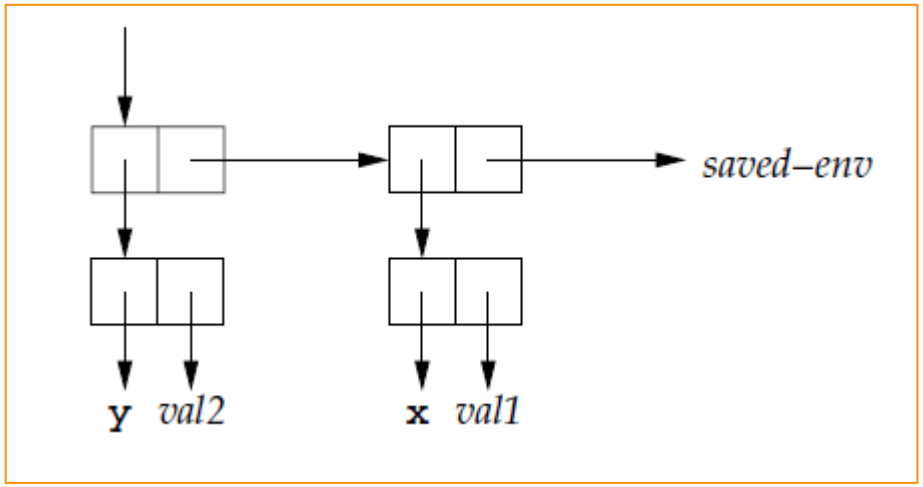
```
let x = exp1
in let y = exp2
   in -(x,y)
```

in our language. In the difference expression, the lexical depths of y and x are 0 and 1, respectively.

Now assume that the values of exp_1 and exp_2, in the appropriate environments, are val_1 and val_2. Then the value of this expression is

...

```
(value-of
  <<let x = exp1
    in let y = exp2
       in -(x,y)>>
  ρ)
=
(value-of
  <<let y = exp2
    in -(x,y)>>
  [x=val1] ρ)
=
(value-of
  <<-(x,y)>>
  [y=val2] [x=val1] ρ)
```



...

The value of this expression is

```
(value-of
  <<let a = 5 in proc (x) -(x,a)>>
  ρ)
= (value-of <<proc (x) -(x,a)>>
    (extend-env a ⌈5⌉ ρ))
= (proc-val (procedure x <<-(x,a)>> [a=⌈5⌉]ρ))
```

The body of this procedure can only be evaluated by `apply-procedure`:

```
(apply-procedure
  (procedure x <<-(x,a)>> [a=⌈5⌉]ρ)
  ⌈7⌉)
= (value-of <<-(x,a)>>
    [x=⌈7⌉] [a=⌈5⌉]ρ)
```

So again every variable is found in the environment at the place predicted by its lexical depth.

Implementing Lexical Addressing

We now implement the lexical-address analysis we sketched above. We write a procedure `translation-of-program` that takes a program and removes all the variables from the declarations, and replaces every variable reference by its lexical depth.

We then write a new version of `value-of-program` that will find the value of such a nameless program, without putting variables in the environment.

...

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

```
#(struct:a-program
   #(struct:nameless-let-exp
      #(struct:const-exp 37)
      #(struct:nameless-proc-exp
         #(struct:nameless-let-exp
            #(struct:diff-exp
               #(struct:nameless-var-exp 0)
               #(struct:nameless-var-exp 1))
            #(struct:diff-exp
               #(struct:nameless-var-exp 2)
               #(struct:nameless-var-exp 1))))))
```

The Translator

We are writing a translator, so we need to know the source language and the target language. The target language will have things like `nameless-var-exp` and `nameless-let-exp` that were not in the source language, and it will lose the things in the source language that these constructs replace, like `var-exp` and `let-exp`.

Expression ::= `%lexref` *number*

```
nameless-var-exp (num)
```

Expression ::= `%let` *Expression* `in` *Expression*

```
nameless-let-exp (exp1 body)
```

Expression ::= `%lexproc` *Expression*

```
nameless-proc-exp (body)
```

...

To calculate the lexical address of any variable reference, we need to know the scopes in which it is enclosed. This is *context* information .

So `translation-of` will take two arguments: an expression and a *static environment*. The static environment will be a list of variables, representing the scopes within which the current expression lies. The variable declared in the innermost scope will be the first element of the list.

For example, when we translate the last line of the example above, the static environment should be

```
(z y x)
```

...

```
(z y x)
```

So looking up a variable in the static environment means finding its position in the static environment, which gives a lexical address: looking up `x` will give 2, looking up `y` will give 1, and looking up `z` will give 0.

Entering a new scope will mean adding a new element to the static environment. We introduce a procedure `extend-senv` to do this.

Since the static environment is just a list of variables, these procedures are easy to implement and are shown in figure 3.15.

$Senv = Listof(Sym)$
$Lexaddr = N$

empty-senv : $() \rightarrow Senv$

```
(define empty-senv
  (lambda ()
    '()))
```

extend-senv : $Var \times Senv \rightarrow Senv$

```
(define extend-senv
  (lambda (var senv)
    (cons var senv)))
```

apply-senv : $Senv \times Var \rightarrow Lexaddr$

```
(define apply-senv
  (lambda (senv var)
    (cond
      ((null? senv)
       (report-unbound-var var))
      ((eqv? var (car senv))
       0)
      (else
        (+ 1 (apply-senv (cdr senv) var))))))
```

Figure 3.15 Implementation of static environments

Writing The Translator

1. Every `var-exp` is replaced by a `nameless-var-exp` with the right lexical address, which we compute by calling `apply-senv`.

2. Every `let-exp` is replaced by a `nameless-let-exp`. The right-hand side of the new expression will be the translation of the right-hand side of the old expression. This is in the same scope as the original, so we translate it in the same static environment `senv`. The body of the new expression will be the translation of the body of the old expression. But the body now lies in a new scope, with the additional bound variable *var*. So we translate the body in the static environment `(extend-senv` *var* `senv)`.

3. Every `proc-exp` is replaced by a `nameless-proc-exp`, with the body translated with respect to the new scope, represented by the static environment `(extend-senv` *var* `senv)`.

...

The procedure `translation-of-program` runs `translation-of` in a suitable initial static environment.

translation-of-program : *Program* → *Nameless-program*

```
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))
```

init-senv : () → *Senv*

```
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

translation-of : *Exp* × *Senv* → *Nameless-exp*

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num) (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp)))))
```

Figure 3.16 The lexical-address translator

The Nameless Interpreter

Our top-level procedure will be run:

run : *String* → *ExpVal*

```
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string)))))
```

The Nameless Interpreter

Instead of having full-fledged environments, we will have nameless environments, with the following interface:

nameless-environment?	: $SchemeVal \rightarrow Bool$
empty-nameless-env	: $() \rightarrow Nameless\text{-}env$
extend-nameless-env	: $Expval \times Nameless\text{-}env \rightarrow Nameless\text{-}env$
apply-nameless-env	: $Nameless\text{-}env \times Lexaddr \rightarrow DenVal$

We can implement a nameless environment as a list of denoted values, so that `apply-nameless-env` is simply a call to `list-ref`. The implementation is shown in figure 3.17.

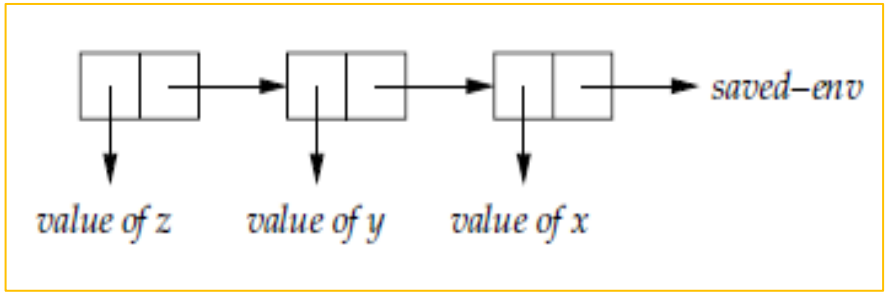


Figure 3.17 Nameless environments

nameless-environment? : *SchemeVal* → *Bool*

```
(define nameless-environment?
  (lambda (x)
    ((list-of expval?) x)))
```

empty-nameless-env : () → *Nameless-env*

```
(define empty-nameless-env
  (lambda ()
    '()))
```

extend-nameless-env : *ExpVal* × *Nameless-env* → *Nameless-env*

```
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))
```

apply-nameless-env : *Nameless-env* × *Lexaddr* → *ExpVal*

```
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

The revised specification for procedures is just the old one with the variable name removed.

```
(apply-procedure (procedure body ρ) val)
= (value-of body (extend-nameless-env val ρ))
```

We can implement this by defining

procedure : *Nameless-exp* × *Nameless-env* → *Proc*

```
(define-datatype proc proc?
  (procedure
    (body expression?)
    (saved-nameless-env nameless-environment?)))
```

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (body saved-nameless-env)
        (value-of body
          (extend-nameless-env val saved-nameless-env))))))
```

...

Now we can write `value-of`. Most cases are the same as in the earlier interpreters except that where we used `env` we now use `nameless-env`. We do have new cases, however, that correspond to `var-exp`, `let-exp`, and `proc-exp`, which we replace by cases for `nameless-var-exp`, `nameless-let-exp`, and `nameless-proc-exp`, respectively. The implementation is shown in figure 3.18. A `nameless-var-exp` gets looked up in the environment. A `nameless-let-exp` evaluates its right-hand side exp_1, and then evalutes its body in an environment extended by the value of the right-hand side. This is just what an ordinary `let` does, but without the variables. A `nameless-proc` produces a `proc`, which is then applied by `apply-procedure`.

value-of : *Nameless-exp* × *Nameless-env* → *ExpVal*

Figure 3.18 `value-of` for the nameless interpreter

```
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num)     ...as before...)
      (diff-exp (exp1 exp2)   ...as before...)
      (zero?-exp (exp1)     ...as before...)
      (if-exp (exp1 exp2 exp3) ...as before...)
      (call-exp (rator rand)   ...as before...)

      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp)))))
```

States

Essentials of Programming Languages (Chapter 4)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Computational Effects

So far, we have only considered the *value* produced by a computation. But a computation may have *effects* as well: it may read, print, or alter the state of memory or a file system.

What's the difference between producing a value and producing an effect? An effect is *global*: it is seen by the entire computation. An effect affects the entire computation

We will be concerned primarily with a single effect: assignment to a location in memory. How does assignment differ from binding? As we have seen, binding is local, but variable assignment is potentially global. It is about the *sharing* of values between otherwise unrelated portions of the computation. Two procedures can share information if they both know about the same location in memory. A single procedure can share information with a future invocation of itself by leaving the information in a known location.

Memory

We model memory as a finite map from *locations* to a set of values called the *storable values*. For historical reasons, we call this the *store*.

A data structure that represents a location is called a *reference*. A location is a place in memory where a value can be stored, and a reference is a data structure that refers to that place. The distinction between locations and references may be seen by analogy: a location is like a file and a reference is like a URL. The URL refers to the file, and the file contains some data. Similarly, a reference denotes a location, and the location contains some data.

. . .

References are sometimes called *L-values*. This name reflects the association of such data structures with variables appearing on the left-hand side of assignment statements. Analogously, expressed values, such as the values of the right-hand side expressions of assignment statements, are known as *R-values*.

We consider two designs for a language with a store. We call these designs *explicit references* and *implicit references*.

EXPLICIT-REFS: A Language with Explicit References

In this design, we add references as a new kind of expressed value. So we have

$$\begin{aligned} ExpVal &= Int + Bool + Proc + Ref(ExpVal) \\ DenVal &= ExpVal \end{aligned}$$

Here $Ref(ExpVal)$ means the set of references to locations that contain expressed values.

We leave the binding structures of the language unchanged, but we add three new operations to create and use references.

- `newref`, which allocates a new location and returns a reference to it.
- `deref`, which dereferences a reference: that is, it returns the contents of the location that the reference represents.
- `setref`, which changes the contents of the location that the reference represents.

EXPLICIT-REFS: Example1

```
let x = newref(0)
in letrec even(dummy)
           = if zero?(deref(x))
             then 1
             else begin
                   setref(x, -(deref(x),1));
                   (odd 888)
                  end
          odd(dummy)
           = if zero?(deref(x))
             then 0
             else begin
                   setref(x, -(deref(x),1));
                   (even 888)
                  end
   in begin setref(x,13); (odd 888) end
```

...

`even` and `odd` each take an argument, which they ignore, and return 1 or 0 depending on whether the contents of the location `x` is even or odd. They communicate not by passing data explicitly, but by changing the contents of the variable they share.

This program determines whether or not 13 is odd, and therefore returns 1. The procedures `even` and `odd` do not refer to their arguments; instead they look at the contents of the location to which `x` is bound.

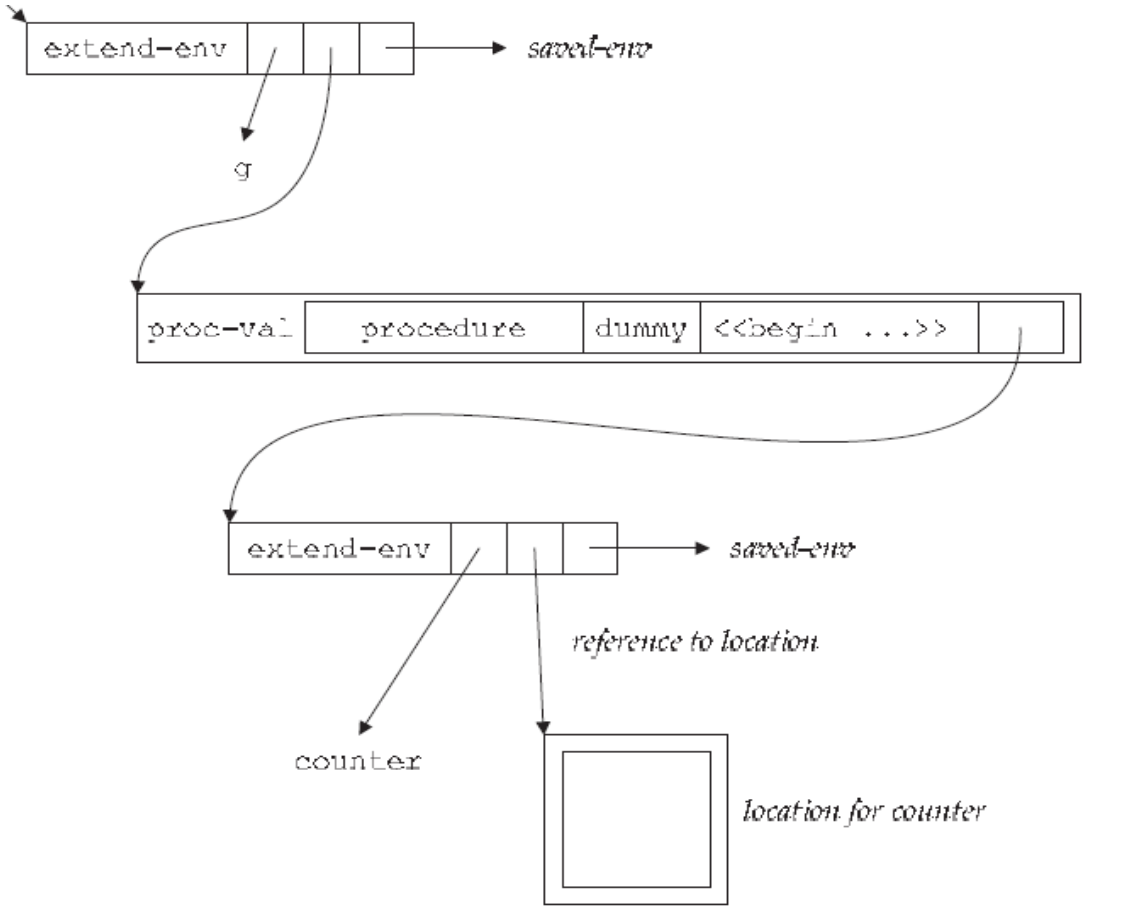
This program uses multideclaration `letrec` (exercise 3.32) and a `begin` expression (exercise 4.4). A `begin` expression evaluates its subexpressions in order and returns the value of the last one.

EXPLICIT-REFS: Example2

```
let g = let counter = newref(0)
        in proc (dummy)
            begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Here the procedure g keeps a private variable that stores the number of times g has been called. Hence the first call to g returns 1, the second call to g returns 2, and the entire program has the value -1.

Example2: a picture of the environment in which g is bound



EXPLICIT-REFS: Example3

In EXPLICIT-REFS, we can store any expressed value, and references are expressed values. This means we can store a reference in a location. Consider the program

```
let x = newref(newref(0))
in begin
    setref(deref(x), 11);
    deref(deref(x))
   end
```

...

This program allocates a new location containing 0. It then binds `x` to a location containing a reference to the first location. Hence the value of `deref(x)` is a reference to the first location. So when the program evaluates the `setref`, it is the first location that is modified, and the entire program returns 11.

EXPLICIT-REFS: Store-Passing Specifications

In our language, any expression may have an effect. To specify these effects, we need to describe what store should be used for each evaluation and how each evaluation can modify the store.

In our specifications, we use σ to range over stores. We write $[l = v]\sigma$ to mean a store just like σ, except that location l is mapped to v. When we refer to a particular value of σ, we sometimes call it the *state* of the store.

EXPLICIT-REFS: Store-Passing Specifications

We use *store-passing specfications*. In a store-passing specification, the store is passed as an explicit argument to `value-of` and is returned as an explicit result from `value-of`. Thus we write

$$(\texttt{value-of}\ \ exp_1\ \ \rho\ \ \sigma_0) = (val_1, \sigma_1)$$

This asserts that expression exp_1, evaluated in environment ρ and with the store in state σ_0, returns the value val_1 and leaves the store in a possibly different state σ_1.

Thus we can specify an effect-free operation like `const-exp` by writing

$$\texttt{(value-of (const-exp } n\texttt{)} \ \rho \ \sigma\texttt{)} = (n, \sigma)$$

showing that the store is unchanged by evaluation of this expression.

The specification for `diff-exp` shows how we specify sequential behavior.

$$\frac{\begin{array}{c}\texttt{(value-of } exp_1 \ \rho \ \sigma_0\texttt{)} = (val_1, \sigma_1) \\ \texttt{(value-of } exp_2 \ \rho \ \sigma_1\texttt{)} = (val_2, \sigma_2)\end{array}}{\texttt{(value-of (diff-exp } exp_1 \ exp_2\texttt{)} \ \rho \ \sigma_0\texttt{)} = (\lceil \lfloor val_1 \rfloor \texttt{-} \lfloor val_2 \rfloor \rceil, \sigma_2)}$$

Here we evaluate exp_1 starting with the store in state σ_0. exp_1 returns value val_1, but it might also have some effects that leave the store in state σ_1. We then evaluate exp_2 starting with the store in the state that exp_1 left it, namely σ_1. exp_2 similarly returns a value val_2 and leaves the store in state σ_2. Then the entire expression returns $val_1 - val_2$ without further effect on the store, so it leaves the store in state σ_2.

Let's try a conditional.

$$\frac{\texttt{(value-of } exp_1 \ \rho \ \sigma_0\texttt{)} = (val_1, \sigma_1)}{\begin{array}{l}\texttt{(value-of (if-exp } exp_1 \ exp_2 \ exp_3\texttt{)} \ \rho \ \sigma_0\texttt{)} \\ \quad = \begin{cases} \texttt{(value-of } exp_2 \ \rho \ \sigma_1\texttt{)} & \text{if } (\texttt{expval->bool } val_1) = \texttt{\#t} \\ \texttt{(value-of } exp_3 \ \rho \ \sigma_1\texttt{)} & \text{if } (\texttt{expval->bool } val_1) = \texttt{\#f} \end{cases}\end{array}}$$

Starting in state σ_0, an `if-exp` evaluates its test expression exp_1, returning the value val_1 and leaving the store in state σ_1. The result of the entire expression is then either the result of exp_2 or exp_3, each evaluated in the current environment ρ and in the state σ_1 in which exp_1 left the store.

Specifying Operations on Explicit References

In EXPLICIT-REFS, we have three new operations that must be specified: `newref`, `deref`, and `setref`. These are given by the grammar

$$\begin{aligned}
\textit{Expression} &::= \texttt{newref}\ \ (\textit{Expression}) \\
&\qquad\ \boxed{\texttt{newref-exp (exp1)}} \\
\textit{Expression} &::= \texttt{deref}\ \ (\textit{Expression}) \\
&\qquad\ \boxed{\texttt{deref-exp (exp1)}} \\
\textit{Expression} &::= \texttt{setref}\ \ (\textit{Expression}\ \ ,\ \ \textit{Expression}) \\
&\qquad\ \boxed{\texttt{setref-exp (exp1 exp2)}}
\end{aligned}$$

...

We can specify the behavior of these operations as follows.

$$\frac{\texttt{(value-of}\ \mathit{exp}\ \rho\ \sigma_0\texttt{)} = \texttt{(}\mathit{val}\texttt{,}\sigma_1\texttt{)} \qquad l \notin \mathrm{dom}(\sigma_1)}{\texttt{(value-of (newref-exp}\ \mathit{exp}\texttt{)}\ \rho\ \sigma_0\texttt{)} = \texttt{((ref-val}\ l\texttt{),[}l\texttt{=}\mathit{val}\texttt{]}\sigma_1\texttt{)}}$$

This rule says that `newref-exp` evaluates its operand. It extends the resulting store by allocating a new location l and puts the value *val* of its argument in that location. Then it returns a reference to a location l that is new. This means that it is not already in the domain of σ_1.

...

$$\frac{(\texttt{value-of}\ \mathit{exp}\ \rho\ \sigma_0) = (l, \sigma_1)}{(\texttt{value-of}\ (\texttt{deref-exp}\ \mathit{exp})\ \rho\ \sigma_0) = (\sigma_1(l), \sigma_1)}$$

This rule says that a `deref-exp` evaluates its operand, leaving the store in state σ_1. The value of that argument should be a reference to a location l. The `deref-exp` then returns the contents of l in σ_1, without any further change to the store.

...

$$\frac{\begin{array}{c}(\texttt{value-of}\ \ exp_1\ \ \rho\ \ \sigma_0) = (l\,,\sigma_1)\\ (\texttt{value-of}\ \ exp_2\ \ \rho\ \ \sigma_1) = (val\,,\sigma_2)\end{array}}{(\texttt{value-of}\ \ (\texttt{setref-exp}\ \ exp_1\ \ exp_2)\ \ \rho\ \ \sigma_0) = (\lceil 23 \rceil\,,\ [l{=}val]\,\sigma_2)}$$

This rule says that a `setref-exp` evaluates its operands from left to right. The value of the first operand must be a reference to a location l. The `setref-exp` then updates the resulting store by putting the value val of the second argument in location l. What should a `setref-exp` return? It could return anything. To emphasize the arbitrary nature of this choice, we have specified that it returns 23. Because we are not interested in the value returned by a `setref-exp`, we say that this expression is executed *for effect*, rather than for its value.

EXPLICIT-REFS: Implementation

We represent the state of the store as a Scheme value, but we do not explicitly pass and return it, as the specification suggests. Instead, we keep the state in a single global variable, to which all the procedures of the implementation have access.

We still have to choose how to model the store as a Scheme value. We choose the simplest possible model: we represent the store as a list of expressed values, and a reference is a number that denotes a position in the list. A new reference is allocated by appending a new value to the list; and updating the store is modeled by copying over as much of the list as necessary. The code is shown in figures 4.1

empty-store : () → *Sto*

```
(define empty-store
  (lambda () '()))
```

usage: A Scheme variable containing the current state of the store. Initially set to a dummy value.

```
(define the-store 'uninitialized)
```

get-store : () → *Sto*

```
(define get-store
  (lambda () the-store))
```

initialize-store! : () → *Unspecified*

usage: (initialize-store!) sets the-store to the empty store

```
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))
```

Figure 4.1 A naive model of the store

reference? : *SchemeVal* → *Bool*

```
(define reference?
  (lambda (v)
    (integer? v)))
```

newref : *ExpVal* → *Ref*

```
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref)))
```

deref : *Ref* → *ExpVal*

```
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

Figure 4.1 A naive model of the store

setref! : *Ref* × *ExpVal* → *Unspecified*

usage: sets the-store to a state like the original, but with position ref containing val.

```
(define setref!
  (lambda (ref val)
    (set! the-store
      (letrec
        ((setref-inner
           usage: returns a list like store1, except that
           position ref1 contains val.
           (lambda (store1 ref1)
             (cond
               ((null? store1)
                (report-invalid-reference ref the-store))
               ((zero? ref1)
                (cons val (cdr store1)))
               (else
                (cons
                  (car store1)
                  (setref-inner
                    (cdr store1) (- ref1 1)))))))
        (setref-inner the-store ref)))))
```

Figure 4.1 A naive model of the store

EXPLICIT-REFS: Implementation

We add a new variant, `ref-val`, to the data type for expressed values, and we modify `value-of-program` to initialize the store before each evaluation.

value-of-program : *Program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (initialize-store!)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

EXPLICIT-REFS: Implementation

Now we can write clauses in `value-of` for `newref`, `deref`, and `setref`.

```
(newref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (ref-val (newref v1))))

(deref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (let ((ref1 (expval->ref v1)))
      (deref ref1))))

(setref-exp (exp1 exp2)
  (let ((ref (expval->ref (value-of exp1 env))))
    (let ((val2 (value-of exp2 env)))
      (begin
        (setref! ref val2)
        (num-val 23)))))
```

Figure 4.4 Trace of an evaluation in EXPLICIT-REFS.

```
> (run "
let x = newref(22)
in let f = proc (z) let zz = newref(-(z,deref(x)))
                    in deref(zz)
   in -((f 66), (f 55))")

entering let x
newref: allocating location 0
entering body of let x with env =
((x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering let f
entering body of let f with env =
((f
  (procedure
    z
    ...
    ((x #(struct:ref-val 0))
     (i #(struct:num-val 1))
     (v #(struct:num-val 5))
     (x #(struct:num-val 10)))))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))
```

```
entering body of proc z with env =
((z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering let zz
newref: allocating location 1
entering body of let zz with env =
((zz #(struct:ref-val 1))
 (z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))
```

Figure 4.4 Trace of an evaluation in EXPLICIT-REFS.

```
entering body of proc z with env =
((z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

entering let zz
newref: allocating location 2
entering body of let zz with env =
((zz #(struct:ref-val 2))
 (z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22))
 (1 #(struct:num-val 44))
 (2 #(struct:num-val 33)))

#(struct:num-val 11)
>
```

IMPLICIT-REFS: A Language with Implicit References

In this design, every variable denotes a reference. Denoted values are references to locations that contain expressed values. References are no longer expressed values. They exist only as the bindings of variables.

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$

Locations are created with each binding operation: at each procedure call, `let`, or `letrec`.

When a variable appears in an expression, we first look up the identifier in the environment to find the location to which it is bound, and then we look up in the store to find the value at that location.

IMPLICIT-REFS: A Language with Implicit References

When a variable appears in an expression, we first look up the identifier in the environment to find the location to which it is bound, and then we look up in the store to find the value at that location. Hence we have a "two-level" system for `var-exp`.

The contents of a location can be changed by a `set` expression. We use the syntax

Expression ::= `set` *Identifier* `=` *Expression*

```
assign-exp (var exp1)
```

Here the *Identifier* is not part of an expression, so it does not get dereferenced. In this design, we say that variables are *mutable*, meaning changeable.

```
let x = 0
in letrec even(dummy)
           = if zero?(x)
             then 1
             else begin
                   set x = -(x,1);
                   (odd 888)
                  end
          odd(dummy)
           = if zero?(x)
             then 0
             else begin
                   set x = -(x,1);
                   (even 888)
                  end
   in begin set x = 13; (odd -888) end

let g = let count = 0
        in proc (dummy)
            begin
             set count = -(count,-1);
             count
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Figure 4.7 odd and even in IMPLICIT-REFS

IMPLICIT-REFS

This design is called *call-by-value*, or *implicit references*.

Because references are no longer expressed values, we can't make chains of references, as we did in the last example in section 4.2.

IMPLICIT-REFS: Specification

We can write the rules for dereference and `set` easily. The environment now always binds variables to locations, so when a variable appears as an expression, we need to dereference it:

$$\texttt{(value-of (var-exp } var\texttt{)} \;\; \rho \;\; \sigma\texttt{)} = (\sigma(\rho(var)), \sigma)$$

IMPLICIT-REFS: Specification

Assignment works as one might expect: we look up the left-hand side in the environment, getting a location, we evaluate the right-hand side in the environment, and we modify the desired location. As with `setref`, the value returned by a `set` expression is arbitrary. We choose to have it return the expressed value 27.

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho\ \ \sigma_0) = (val_1, \sigma_1)}{(\texttt{value-of}\ \ (\texttt{assign-exp}\ \ var\ \ exp_1)\ \ \rho\ \ \sigma_0) = (\lceil \texttt{27} \rceil, [\rho(var) = val_1]\sigma_1)}$$

IMPLICIT-REFS: Specification

We also need to rewrite the rules for procedure call and `let` to show the modified store. For procedure call, the rule becomes

$$\texttt{(apply-procedure (procedure } var\ body\ \rho\texttt{)}\ val\ \sigma\texttt{)}$$
$$= \texttt{(value-of } body\ [var = l]\rho\ [l = val]\sigma\texttt{)}$$

where l is a location not in the domain of σ.

The rule for `(let-exp` $var\ exp_1\ body$`)` is similar. The right-hand side exp_1 is evaluated, and the value of the `let` expression is the value of the body, evaluated in an environment where the variable var is bound to a new location containing the value of exp_1.

IMPLICIT-REFS: The Implementation

Now we are ready to modify the interpreter. In `value-of`, we dereference at each `var-exp`, just like the rules say

```
(var-exp (var) (deref (apply-env env var)))
```

and we write the obvious code for a `assign-exp`

```
(assign-exp (var exp1)
  (begin
    (setref!
      (apply-env env var)
      (value-of exp1 env))
    (num-val 27)))
```

IMPLICIT-REFS: The Implementation

What about creating references? New locations should be allocated at every new binding. There are exactly four places in the language where new bindings are created: in the initial environment, in a `let`, in a procedure call, and in a `letrec`.

In the initial environment, we explicitly allocate new locations.

For `let`, we change the corresponding line in `value-of` to allocate a new location containing the value, and to bind the variable to a reference to that location.

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var (newref val1) env))))
```

IMPLICIT-REFS: The Implementation

For a procedure call, we similarly change `apply-procedure` to call `newref`.

apply-procedure : $Proc \times ExpVal \rightarrow ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

Figure 4.8 Sample evaluation in IMPLICIT-REFS

```
> (run "
let f = proc (x) proc (y)
        begin
         set x = -(x,-1);
         -(x,y)
        end
in ((f 44) 33)")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))
```

```
newref: allocating location 4
entering body of proc x with env =
((x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44)))

newref: allocating location 5
entering body of proc y with env =
((y 5) (x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44))
 (5 #(struct:num-val 33)))

#(struct:num-val 12)
```

MUTABLE-PAIRS: A Language with Mutable Pairs

Now, let's add mutable pairs to IMPLICIT-REFS. Pairs will be expressed values, and will have the following operations:

newpair	$: Expval \times Expval \rightarrow MutPair$
left	$: MutPair \rightarrow Expval$
right	$: MutPair \rightarrow Expval$
setleft	$: MutPair \times Expval \rightarrow Unspecified$
setright	$: MutPair \times Expval \rightarrow Unspecified$

A pair consists of two locations, each of which is independently assignable. This gives us the domain equations:

$$\begin{aligned} ExpVal &= Int + Bool + Proc + MutPair \\ DenVal &= Ref(ExpVal) \\ MutPair &= Ref(ExpVal) \times Ref(ExpVal) \end{aligned}$$

We call this language MUTABLE-PAIRS.

MUTABLE-PAIRS: Implementation

We add a `mutpair-val` variant to our data type of expressed values, and five new lines to `value-of`. These are shown in figure 4.10. We arbitrarily choose to make `setleft` return 82 and `setright` return 83. The trace of an example, using the same instrumentation as before, is shown in figures 4.11 and 4.12.

Figure 4.9 Naive implementation of mutable pairs

```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))
```

make-pair : *ExpVal* × *ExpVal* → *MutPair*

```
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))
```

left : *MutPair* → *ExpVal*

```
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))
```

right : *MutPair* → *ExpVal*

```
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc)))))
```

setleft : *MutPair* × *ExpVal* → *Unspecified*

```
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val)))))
```

setright : *MutPair* × *ExpVal* → *Unspecified*

```
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))
```

Figure 4.10 Integrating mutable pairs into the interpreter

```
(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (left p1))))

(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1))))

(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))

(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))
```

Figure 4.11 Trace of evaluation in MUTABLE-PAIRS

```
> (run "let glo = pair(11,22)
in let f = proc (loc)
            let d1 = setright(loc, left(loc))
            in let d2 = setleft(glo, 99)
            in -(left(loc),right(loc))
in (f glo)")
;; allocating cells for init-env
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let glo
;; allocating cells for the pair
newref: allocating location 3
newref: allocating location 4
;; allocating cell for glo
newref: allocating location 5
entering body of let glo with env =
((glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4))))
```

```
entering let f
;; allocating cell for f
newref: allocating location 6
entering body of let f with env =
((f 6) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2)))))
;; allocating cell for loc
newref: allocating location 7
entering body of proc loc with env =
((loc 7) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2))))
 (7 #(struct:mutpair-val #(struct:a-pair 3 4))))

#(struct:num-val 88)
>
```

Another Representation of Mutable Pairs

The representation of a mutable pair as two references does not take advantage of all we know about *MutPair*. The two locations in a pair are independently assignable, but they are not independently allocated. We know that they will be allocated together: if the left part of a pair is one location, then the right part is in the next location. So we can instead represent the pair by a reference to its left. The code for this is shown in figure 4.13. Nothing else need change.

Figure 4.13 Alternate representation of mutable pairs

mutpair? : *SchemeVal* → *Bool*

```
(define mutpair?
  (lambda (v)
    (reference? v)))
```

make-pair : *ExpVal* × *ExpVal* → *MutPair*

```
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1))))
```

left : *MutPair* → *ExpVal*

```
(define left
  (lambda (p)
    (deref p)))
```

right : *MutPair* → *ExpVal*

```
(define right
  (lambda (p)
    (deref (+ 1 p))))
```

setleft : *MutPair* × *ExpVal* → *Unspecified*

```
(define setleft
  (lambda (p val)
    (setref! p val)))
```

setright : *MutPair* × *ExpVal* → *Unspecified*

```
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

...

Similarly, one could represent any aggregate object in the heap by a pointer to its first location. However, a pointer does not by itself identify an area of memory unless it is supplemented by information about the length of the area (see exercise 4.30). The lack of length information is a source of classic security errors, such as out-of-bounds array writes.

Parameter-Passing Variations

When a procedure body is executed, its formal parameter is bound to a denoted value. Where does that value come from? It must be passed from the actual parameter in the procedure call. We have already seen two ways in which a parameter can be passed:

- Natural parameter passing, in which the denoted value is the same as the expressed value of the actual parameter (page 75).
- Call-by-value, in which the denoted value is a reference to a location containing the expressed value of the actual parameter (section 4.3). This is the most commonly used form of parameter-passing.

In this section, we explore some alternative parameter-passing mechanisms.

Parameter-Passing Variations: CALL-BY-REFERENCE

Consider the following expression:

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Under call-by-value, the denoted value associated with x is a reference that initially contains the same value as the reference associated with a, but these references are distinct. Thus the assignment to x has no effect on the contents of a's reference, so the value of the entire expression is 3.

With call-by-value, when a procedure assigns a new value to one of its parameters, this cannot possibly be seen by its caller.

Parameter-Passing Variations: CALL-BY-REFERENCE

Though this isolation between the caller and callee is generally desirable, there are times when it is valuable to allow a procedure to be passed locations with the expectation that they will be assigned by the procedure. This may be accomplished by passing the procedure a reference to the location of the caller's variable, rather than the contents of the variable. This parameter-passing mechanism is called *call-by-reference*. If an operand is simply a variable reference, a reference to the variable's location is passed. The formal parameter of the procedure is then bound to this location. If the operand is some other kind of expression, then the formal parameter is bound to a new location containing the value of the operand, just as in call-by-value.

Parameter-Passing Variations: CALL-BY-REFERENCE

When a call-by-reference procedure is called and the actual parameter is a variable, what is passed is the *location* of that variable, rather than the contents of that location, as in call-by-value. For example, consider

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin (g z); z end
```

When the procedure `g` is called, `y` is bound to the location of `z`, not the contents of that location. Similarly, when `f` is called, `x` becomes bound to that same location. So `x`, `y`, and `z` will all be bound to the same location, and the effect of the `set x = 44` is to set that location to 44. Hence the value of the entire expression is 44.

...

A typical use of call-by-reference is to return multiple values. A procedure can return one value in the normal way and assign others to parameters that are passed by reference. For another sort of example, consider the problem of swapping the values in two variables:

```
let swap = proc (x) proc (y)
            let temp = x
            in begin
                set x = y;
                set y = temp
               end
in let a = 33
   in let b = 44
      in begin
          ((swap a) b);
          -(a,b)
         end
```

...

Under call-by-reference, this swaps the values of `a` and `b`, so it returns 11. If this program were run with our existing call-by-value interpreter, however, it would return -11, because the assignments inside the swap procedure then have no effect on variables `a` and `b`.

CALL-BY-REFERENCE

Under call-by-reference, variables still denote references to expressed values, just as they did under call-by-value:

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$

The only thing that changes is the allocation of new locations. Under call-by-value, a new location is created for every evaluation of an operand; under call-by-reference, a new location is created for every evaluation of an operand *other than a variable*.

CALL-BY-REFERENCE: Implementation

We modify the `call-exp` line in `value-of`, and introduce a new function `value-of-operand` that makes the necessary decision.

```
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of-operand rand env)))
    (apply-procedure proc arg)))
```

The procedure `value-of-operand` checks to see if the operand is a variable. If it is, then the reference that the variable denotes is returned and then passed to the procedure by `apply-procedure`. Otherwise, the operand is evaluated, and a reference to a new location containing that value is returned.

CALL-BY-REFERENCE: Implementation

apply-procedure : $Proc \times Ref \rightarrow ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var val saved-env))))))
```

value-of-operand : $Exp \times Env \rightarrow Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (value-of exp env))))))
```

CALL-BY-REFERENCE: Aliases

More than one call-by-reference parameter may refer to the same location, as in the following program.

```
let b = 3
in let p = proc (x) proc(y)
            begin
             set x = 4;
             y
            end
   in ((p b) b)
```

This yields 4 since both x and y refer to the same location, which is the binding of b. This phenomenon is known as *variable aliasing*. Here x and y are aliases (names) for the same location. Generally, we do not expect an assignment to one variable to change the value of another, so aliasing makes it very difficult to understand programs.

Figure 4.14 Sample evaluation in CALL-BY-REFERENCE

```
>  (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
    (g z);
    z
  end")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

entering let g
newref: allocating location 4
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))
```

```
entering let z
newref: allocating location 5
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

 entering body of proc y with env =
 ((y 5) (f 3) (i 0) (v 1) (x 2))
 store =
 ((0 #(struct:num-val 1))
  (1 #(struct:num-val 5))
  (2 #(struct:num-val 10))
  (3 (procedure x ... ((i 0) (v 1) (x 2))))
  (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
  (5 #(struct:num-val 55)))

 entering body of proc x with env =
 ((x 5) (i 0) (v 1) (x 2))
 store =
 ((0 #(struct:num-val 1))
  (1 #(struct:num-val 5))
  (2 #(struct:num-val 10))
  (3 (procedure x ... ((i 0) (v 1) (x 2))))
  (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
  (5 #(struct:num-val 55)))

 #(struct:num-val 44)
 >
```

Lazy Evaluation: CALL-BY-NAME and CALL-BY-NEED

All the parameter-passing mechanisms we have discussed so far are *eager*: they always find a value for each operand. We now turn to a very different form of parameter passing, called *lazy evaluation*. Under lazy evaluation, an operand in a procedure call is not evaluated until it is needed by the procedure body. If the body never refers to the parameter, then there is no need to evaluate it.

Lazy Evaluation: CALL-BY-NAME and CALL-BY-NEED

This can potentially avoid non-termination. For example, consider

```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

Here `infinite-loop` is a procedure that, when called, never terminates. `f` is a procedure that, when called, never refers to its argument and always returns 11. Under any of the mechanisms considered so far, this program will fail to terminate. Under lazy evaluation, however, this program will return 11, because the operand `(infinite-loop 1)` is never evaluated.

...

We now modify our language to use lazy evaluation. Under lazy evaluation, we do not evaluate an operand expression until it is needed. Therefore we associate the bound variable of a procedure with an unevaluated operand. When the procedure body needs the value of its bound variable, the associated operand is evaluated. We sometimes say that the operand is *frozen* when it is passed unevaluated to the procedure, and that it is *thawed* when the procedure evaluates it.

...

Of course we will also have to include the environment in which that procedure is to be evaluated. To do this, we introduce a new data type of *thunks*. A thunk consists of an expression and an environment.

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

When a procedure needs to use the value of its bound variable, it will evaluate the associated thunk.

...

We therefore let our denoted values be references to locations containing either expressed values or thunks.

$$
\begin{aligned}
DenVal &= Ref(ExpVal + Thunk) \\
ExpVal &= Int + Bool + Proc
\end{aligned}
$$

Our policy for allocating new locations will be similar to the one we used for call-by-reference: If the operand is a variable, then we pass its denotation, which is a reference. Otherwise, we pass a reference to a new location containing a thunk for the unevaluated argument.

...

value-of-operand : $Exp \times Env \rightarrow Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (a-thunk exp env))))))
```

When we evaluate a `var-exp`, we first find the location to which the variable is bound. If the location contains an expressed value, then that value is returned as the value of the `var-exp`. If it instead contains a thunk, then the thunk is evaluated, and that value is returned. This design is called *call by name*.

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (value-of-thunk w)))))
```

The procedure `value-of-thunk` is defined as

value-of-thunk : *Thunk* → *ExpVal*

```
(define value-of-thunk
  (lambda (th)
    (cases thunk th
      (a-thunk (exp1 saved-env)
        (value-of exp1 saved-env)))))
```

...

Alternatively, once we find the value of the thunk, we can install that expressed value in the same location, so that the thunk will not be evaluated again. This arrangement is called *call by need*.

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (let ((val1 (value-of-thunk w)))
          (begin
            (setref! ref1 val1)
            val1))))))
```

Types

Essentials of Programming Languages (Chapter 7)

BY: MOHAMMAD IZADI

Program Safety Analysis

We've seen how we can use interpreters to model the run-time behavior of programs. Now we'd like to use the same technology to *analyze* or *predict* the behavior of programs without running them.

Our goal is to analyze a program to predict whether evaluation of a program is *safe*, that is, whether the evaluation will proceed without certain kinds of errors. Exactly what is meant by safety, however, may vary from language to language. If we can guarantee that evaluation is safe, we will be sure that the program satisfies its contract.

Safety in LETREC

In this chapter, we will consider languages that are similar to LETREC in chapter 3. For these languages we say that an evaluation is *safe* if and only if:

1. For every evaluation of a variable *var*, the variable is bound.
2. For every evaluation of a difference expression `(diff-exp` exp_1 exp_2`)`, the values of exp_1 and exp_2 are both `num-val`s.
3. For every evaluation of an expression of the form `(zero?-exp` exp_1`)`, the value of exp_1 is a `num-val`.
4. For every evaluation of a conditional expression `(if-exp` exp_1 exp_2 exp_3`)`, the value of exp_1 is a `bool-val`.
5. For every evaluation of a procedure call `(call-exp` *rator* *rand*`)`, the value of *rator* is a `proc-val`.

Some Notes

- We call violations of these type safety conditions *type errors*.
- A safe evaluation may still fail for other reasons: division by zero, taking the car of an empty list, etc. We do not include these as part of our definition of safety.
- We do not include non-termination as part of safety because checking for termination is also very difficult (indeed, it is undecidable in general).
- Our goal is to write a procedure that looks at the program text and either accepts or rejects it. Furthermore, we would like our analysis procedure to be conservative: if the analysis accepts the program, then we can be sure evaluation of the program will be safe. If the analysis cannot be sure that evaluation will be safe, it must reject the program. In this case, we say that the analysis is *sound*.

Examples of type analysis

`if 3 then 88 else 99`	reject: non-boolean test
`proc (x) (3 x)`	reject: non-proc-val rator
`proc (x) (x 3)`	accept
`proc (f) proc (x) (f x)`	accept
`let x = 4 in (x 3)`	reject: non-proc-val rator
`(proc (x) (x 3)` `4)`	reject: same as preceding example
`let x = zero?(0)` `in -(3, x)`	reject: non-integer argument to a diff-exp
`(proc (x) -(3,x)` `zero?(0))`	reject: same as preceding example
`let f = 3` `in proc (x) (f x)`	reject: non-proc-val rator
`(proc (f) proc (x) (f x)` `3)`	reject: same as preceding example
`letrec f(x) = (f -(x,-1))` `in (f 1)`	accept nonterminating but safe

Values and Their Types

- Since the safety conditions talk only about *num-val, bool-val*, and *proc-val*, one might think that it would be enough to keep track of these three types. But that is not enough: if all we know is that f is bound to a *proc-val*, then we can not draw any conclusions whatsoever about the value of *(f 1)*. We need to keep track of finer information about procedures. This finer information is called the *type structure* of the language.

- Types for LETREC language:

Grammar for Types

Type ::= `int`	
	`int-type ()`
Type ::= `bool`	
	`bool-type ()`
Type ::= `(`*Type* `->` *Type*`)`	
	`proc-type (arg-type result-type)`

Examples of values and their types

- The value of *3* has type *int*.
- The value of *-(33,22)* has type *int*.
- The value of *zero?(11)* has type *bool*.
- The value of *proc (x) -(x,11)* has type *(int -> int)*.
- The value of *proc (x) let y = -(x,11) in -(x,y)* has type *(int -> int)*.
- The value of *proc (x) if x then 11 else 22* has type *(bool -> int)*.
- The value of *proc (x) if x then 11 else zero?(11)* has no type in our type system.

Examples of values and their types

- The value of *proc (x) proc (y) if y then x else 11* has type *(int -> (bool -> int).*
- The value of *proc (f) if (f 3) then 11 else 22* has type ((int -> bool) -> int).
- The value of *proc (f) (f 3)* has type *((int -> t) -> t)* for any type *t*.
- The value of *proc (f) proc (x) (f (f x))* has type *((t -> t) -> (t -> t))* for any type *t*.

Definition of types for values and expressions

Definition 7.1.1 *The property of an expressed value v being of type t is defined by induction on t:*

- *An expressed value is of type* `int` *if and only if it is a* `num-val`.
- *It is of type* `bool` *if and only if it is a* `bool-val`.
- *It is of type* $(t_1 \rightarrow t_2)$ *if and only if it is a* `proc-val` *with the property that if it is given an argument of type t_1, then one of the following things happens:*

  1. *it returns a value of type t_2*
  2. *it fails to terminate*
  3. *it fails with an error other than a type error.*

We occasionally say "v has type t" instead of "v is of type t."

Assigning a Type to an Expression

More precisely, our goal is to write a procedure `type-of` which, given an expression (call it *exp*) and a *type environment* (call it *tenv*) mapping each variable to a type, assigns to *exp* a type *t* with the property that

Specification of `type-of`

Whenever *exp* is evaluated in an environment in which each variable has a value of the type specified for it by *tenv*, one of the following happens:

- the resulting value has type *t*,
- the evaluation does not terminate, or
- the evaluation fails on an error other than a type error.

If we can assign an expression to a type, we say that the expression is *well-typed*; otherwise we say it is *ill-typed* or *has no type*.

Typing Rules

$$\texttt{(type-of (const-exp } num\texttt{) } tenv\texttt{) = int}$$

$$\texttt{(type-of (var-exp } var\texttt{) } tenv\texttt{) = } tenv\texttt{(}var\texttt{)}$$

$$\frac{\texttt{(type-of } exp_1 \texttt{ } tenv\texttt{) = int}}{\texttt{(type-of (zero?-exp } exp_1\texttt{) } tenv\texttt{) = bool}}$$

$$\frac{\texttt{(type-of } exp_1 \texttt{ } tenv\texttt{) = int} \quad \texttt{(type-of } exp_2 \texttt{ } tenv\texttt{) = int}}{\texttt{(type-of (diff-exp } exp_1 \texttt{ } exp_2\texttt{) } tenv\texttt{) = int}}$$

$$\frac{\texttt{(type-of } exp_1 \texttt{ } tenv\texttt{) = } t_1 \quad \texttt{(type-of } body \texttt{ [}var\texttt{=}t_1\texttt{]}tenv\texttt{) = } t_2}{\texttt{(type-of (let-exp } var \texttt{ } exp_1 \texttt{ } body\texttt{) } tenv\texttt{) = } t_2}$$

$$\frac{\begin{array}{l}\texttt{(type-of } exp_1 \texttt{ } tenv\texttt{) = bool} \\ \texttt{(type-of } exp_2 \texttt{ } tenv\texttt{) = } t \\ \texttt{(type-of } exp_3 \texttt{ } tenv\texttt{) = } t\end{array}}{\texttt{(type-of (if-exp } exp_1 \texttt{ } exp_2 \texttt{ } exp_3\texttt{) } tenv\texttt{) = } t}$$

$$\frac{\texttt{(type-of } rator \texttt{ } tenv\texttt{) = (}t_1 \rightarrow t_2\texttt{)} \quad \texttt{(type-of } rand \texttt{ } tenv\texttt{) = } t_1}{\texttt{(type-of (call-exp } rator \texttt{ } rand\texttt{) } tenv\texttt{) = } t_2}$$

$$\frac{\texttt{(type-of } body \texttt{ [}var\texttt{=}t_1\texttt{]}tenv\texttt{) = } t_2}{\texttt{(type-of (proc-exp } var \texttt{ } body\texttt{) } tenv\texttt{) = (}t_1 \rightarrow t_2\texttt{)}}$$

Tow main type system design streams

- *Type Checking*: In this approach the programmer is required to supply the missing information about the types of bound variables, and the type checker deduces the types of the other expressions and checks them for consistency.

- *Type Inference*: In this approach the type-checker attempts to *infer* the types of the bound variables based on how the variables are used in the program. If the language is carefully designed, the type-checker can infer most or all of these types.

CHECKED: A Type-Checked Language

CHECKED is the same as LETREC, except that we require the programmer to include the types of all bound variables. Here are some example programs in CHECKED.

```
proc (x : int) -(x,1)

letrec
 int double (x : int) = if zero?(x)
                        then 0
                        else -((double -(x,1)), -2)
in double

proc (f : (bool -> int)) proc (n : int) (f zero?(n))
```

The result type of `double` is `int`, but the type of `double` itself is `(int -> int)`, since it is a procedure that takes an integer and returns an integer.

CHECKED: Syntax

Changed productions for CHECKED

Expression ::= `proc` (*Identifier* : *Type*) *Expression*

```
proc-exp (var ty body)
```

Expression ::= **letrec**
Type *Identifier* (*Identifier* : *Type*) = *Expression*
`in` *Expression*

```
letrec-exp
  (p-result-type p-name b-var b-var-type
   p-body
   letrec-body)
```

CHECKED: Type Rules

(in addition to rules in slide 11)

$$\frac{(\texttt{type-of}\ \ body\ \ [var{=}t_{var}]\,tenv)\ =\ t_{res}}{(\texttt{type-of}\ \ (\texttt{proc-exp}\ \ var\ \ t_{var}\ \ body)\ \ tenv)\ =\ (t_{var}\ \rightarrow\ t_{res})}$$

$$\frac{\begin{array}{c}(\texttt{type-of}\ \ e_{proc\text{-}body}\ \ [var{=}t_{var}]\,[p = (t_{var}\ \rightarrow\ t_{res})\,]\,tenv)\ =\ t_{res}\\ (\texttt{type-of}\ \ e_{letrec\text{-}body}\ \ [p = (t_{var}\ \rightarrow\ t_{res})\,]\,tenv)\ =\ t\end{array}}{(\texttt{type-of letrec}\ t_{res}\ p\,(var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}\ \ tenv)\ =\ t}$$

CHECKED: Type Checker

check-equal-type! : *Type* × *Type* × *Exp* → *Unspecified*

```
(define check-equal-type!
  (lambda (ty1 ty2 exp)
    (if (not (equal? ty1 ty2))
      (report-unequal-types ty1 ty2 exp))))
```

report-unequal-types : *Type* × *Type* × *Exp* → *Unspecified*

```
(define report-unequal-types
  (lambda (ty1 ty2 exp)
    (eopl:error 'check-equal-type!
      "Types didn't match: ~s != ~a in~%~a"
      (type-to-external-form ty1)
      (type-to-external-form ty2)
      exp)))
```

type-to-external-form : *Type* → *List*

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type))))))
```

CHECKED: Type Checker (cont'd)

$Tenv = Var \rightarrow Type$

type-of-program : $Program \rightarrow Type$

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1) (type-of exp1 (init-tenv))))))
```

type-of : $Exp \times Tenv \rightarrow Type$

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
```

$$\boxed{(\texttt{type-of}\ num\ tenv) = \texttt{int}}$$

```
      (const-exp (num) (int-type))
```

$$\boxed{(\texttt{type-of}\ var\ tenv) = tenv(var)}$$

```
      (var-exp (var) (apply-tenv tenv var))
```

$$\boxed{\frac{(\texttt{type-of}\ e_1\ tenv) = \texttt{int} \qquad (\texttt{type-of}\ e_2\ tenv) = \texttt{int}}{(\texttt{type-of}\ (\texttt{diff-exp}\ e_1\ e_2)\ tenv) = \texttt{int}}}$$

```
      (diff-exp (exp1 exp2)
        (let ((ty1 (type-of exp1 tenv))
              (ty2 (type-of exp2 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (check-equal-type! ty2 (int-type) exp2)
          (int-type)))
```

$$\boxed{\frac{(\texttt{type-of}\ e_1\ tenv) = \texttt{int}}{(\texttt{type-of}\ (\texttt{zero?-exp}\ e_1)\ tenv) = \texttt{bool}}}$$

```
      (zero?-exp (exp1)
        (let ((ty1 (type-of exp1 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (bool-type)))
```

$$\boxed{\frac{(\texttt{type-of}\ e_1\ tenv) = \texttt{bool} \quad (\texttt{type-of}\ e_2\ tenv) = t \quad (\texttt{type-of}\ e_3\ tenv) = t}{(\texttt{type-of}\ (\texttt{if-exp}\ e_1\ e_2\ e_3)\ tenv) = t}}$$

```
      (if-exp (exp1 exp2 exp3)
        (let ((ty1 (type-of exp1 tenv))
              (ty2 (type-of exp2 tenv))
              (ty3 (type-of exp3 tenv)))
          (check-equal-type! ty1 (bool-type) exp1)
          (check-equal-type! ty2 ty3 exp)
          ty2))
```

CHECKED: Type Checker (cont'd)

$$\frac{(\texttt{type-of}\ e_1\ tenv) = t_1 \qquad (\texttt{type-of}\ body\ [var{=}t_1]tenv) = t_2}{(\texttt{type-of}\ (\texttt{let-exp}\ var\ e_1\ body)\ tenv) = t_2}$$

```
(let-exp (var exp1 body)
  (let ((exp1-type (type-of exp1 tenv)))
    (type-of body
      (extend-tenv var exp1-type tenv))))
```

$$\frac{(\texttt{type-of}\ body\ [var{=}t_{var}]tenv) = t_{res}}{(\texttt{type-of}\ (\texttt{proc-exp}\ var\ t_{var}\ body)\ tenv) = (t_{var} \rightarrow t_{res})}$$

```
(proc-exp (var var-type body)
  (let ((result-type
          (type-of body
            (extend-tenv var var-type tenv))))
    (proc-type var-type result-type)))
```

$$\frac{(\texttt{type-of}\ rator\ tenv) = (t_1 \rightarrow t_2) \qquad (\texttt{type-of}\ rand\ tenv) = t_1}{(\texttt{type-of}\ (\texttt{call-exp}\ rator\ rand)\ tenv) = t_2}$$

```
(call-exp (rator rand)
  (let ((rator-type (type-of rator tenv))
        (rand-type  (type-of rand tenv)))
    (cases type rator-type
      (proc-type (arg-type result-type)
        (begin
          (check-equal-type! arg-type rand-type rand)
          result-type))
      (else
        (report-rator-not-a-proc-type
          rator-type rator)))))
```

CHECKED: Type Checker (cont'd)

$$\frac{\texttt{(type-of } e_{proc\text{-}body} \ [var{=}t_{var}]\,[p = (t_{var} \rightarrow t_{res})]\,tenv) = t_{res} \qquad \texttt{(type-of } e_{letrec\text{-}body} \ [p = (t_{var} \rightarrow t_{res})]\,tenv) = t}{\texttt{(type-of letrec } t_{res}\ p\,(var : t_{var}) = e_{proc\text{-}body} \texttt{ in } e_{letrec\text{-}body} \ tenv) = t}$$

```
(letrec-exp (p-result-type p-name b-var b-var-type
             p-body letrec-body)
  (let ((tenv-for-letrec-body
          (extend-tenv p-name
            (proc-type b-var-type p-result-type)
            tenv)))
    (let ((p-body-type
            (type-of p-body
              (extend-tenv b-var b-var-type
                tenv-for-letrec-body))))
      (check-equal-type!
        p-body-type p-result-type p-body)
      (type-of letrec-body tenv-for-letrec-body)))))))
```

INFERRED: A Language with Type Inference

For our case study in type inference, we start with the language of CHECKED. We then change the language so that all the type expressions are optional. In place of a missing type expression, we use the marker ?. Hence a typical program looks like

```
letrec
 ? foo (x : ?) = if zero?(x)
                 then 1
                 else -(x, (foo -(x,1)))
in foo
```

Since the type expressions are optional, we may fill in some of the ?'s with types, as in

```
letrec
 ? even (x : int) = if zero?(x) then 1 else (odd -(x,1))
 bool odd (x : ?) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

INFERRED: Syntax

Optional-type ::= ?

```
no-type ()
```

Optional-type ::= *Type*

```
a-type (ty)
```

Expression ::= `proc` (*Identifier* : *Optional-type*) *Expression*

```
proc-exp (var otype body)
```

Expression ::= `letrec`
Optional-type *Identifier* (*Identifier* : *Optional-type*) = *Expression*
`in` *Expression*

```
letrec-exp
  (p-result-otype p-name
   b-var b-var-otype p-body
   letrec-body)
```

Finding the omitted types over AST

The omitted types will be treated as unknowns that we need to find. We do this by traversing the abstract syntax tree and generating equations between these types, possibly including these unknowns. We then solve the equations for the unknown types.

To see how this works, we need names for the unknown types. For each expression e or bound variable var, let t_e or t_{var} denote the type of the expression or bound variable.

For each node in the abstract syntax tree of the expression, the type rules dictate some equations that must hold between these types. For our PROC language, the equations would be:

Main Type Equations

$$\texttt{(diff-exp } e_1 \ e_2\texttt{)} \ : \ t_{e_1} = \texttt{int}, \quad t_{e_2} = \texttt{int}, \quad t_{\texttt{(diff-exp } e_1 \ e_2\texttt{)}} = \texttt{int}$$

$$\texttt{(zero?-exp } e_1\texttt{)} \ : \ t_{e_1} = \texttt{int}, \quad t_{\texttt{(zero?-exp } e_1\texttt{)}} = \texttt{bool}$$

$$\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)} \ : \ t_{e_1} = \texttt{bool}, \quad t_{e_2} = t_{\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)}}, \quad t_{e_3} = t_{\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)}}$$

$$\texttt{(proc-exp } var \ body\texttt{)} \ : \ t_{\texttt{(proc-exp } var \ body\texttt{)}} = (t_{var} \rightarrow t_{body})$$

$$\texttt{(call-exp } rator \ rand\texttt{)} \ : \ t_{rator} = (t_{rand} \rightarrow t_{\texttt{(call-exp } rator \ rand\texttt{)}})$$

Type Inference: an example

To infer the type of an expression, we'll introduce a type variable for every subexpression and every bound variable, generate the constraints for each subexpression, and then solve the resulting equations. To see how this works, we will infer the types of several sample expressions.

Let us start with the expression `proc (f) proc(x) -((f 3),(f x))`. We begin by making a table of all the bound variables, `proc` expressions, `if` expressions, and procedure calls in this expression, and assigning a type variable to each one.

Expression	Type Variable
`f`	t_f
`x`	t_x
`proc(f)proc(x)-((f 3),(f x))`	t_0
`proc(x)-((f 3),(f x))`	t_1
`-((f 3),(f x))`	t_2
`(f 3)`	t_3
`(f x)`	t_4

Type Inference: an example (cont'd)

Now, for each compound expression, we can write down a type equation according to the rules above.

Expression	Equations
`proc(f)proc(x)-((f 3),(f x))`	1. $t_0 = t_f \rightarrow t_1$
`proc(x)-((f 3),(f x))`	2. $t_1 = t_x \rightarrow t_2$
`-((f 3),(f x))`	3. $t_3 = \texttt{int}$
	4. $t_4 = \texttt{int}$
	5. $t_2 = \texttt{int}$
`(f 3)`	6. $t_f = \texttt{int} \rightarrow t_3$
`(f x)`	7. $t_f = t_x \rightarrow t_4$

Unification, substitution and bounding

We separate the state of our calculation into the set of equations still to be solved and the substitution found so far. Initially, all of the equations are to be solved, and the substitution found is empty.

Equations	**Substitution**
$t_0 = t_f \rightarrow t_1$	
$t_1 = t_x \rightarrow t_2$	
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \rightarrow t_3$	
$t_f = t_x \rightarrow t_4$	

We consider each equation in turn. If the equation's left-hand side is a variable, we move it to the substitution.

Equations	**Substitution**
$t_1 = t_x \rightarrow t_2$	$t_0 = t_f \rightarrow t_1$
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \rightarrow t_3$	
$t_f = t_x \rightarrow t_4$	

Type Inference: an example (cont'd)

Equations
$t_3 = \texttt{int}$
$t_4 = \texttt{int}$
$t_2 = \texttt{int}$
$t_f = \texttt{int} \rightarrow t_3$
$t_f = t_x \rightarrow t_4$

Substitution
$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_1 = t_x \rightarrow t_2$

Equations
$t_4 = \texttt{int}$
$t_2 = \texttt{int}$
$t_f = \texttt{int} \rightarrow t_3$
$t_f = t_x \rightarrow t_4$

Substitution
$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_1 = t_x \rightarrow t_2$
$t_3 = \texttt{int}$

Equations
$t_2 = \texttt{int}$
$t_f = \texttt{int} \rightarrow t_3$
$t_f = t_x \rightarrow t_4$

Substitution
$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_1 = t_x \rightarrow t_2$
$t_3 = \texttt{int}$
$t_4 = \texttt{int}$

Equations
$t_f = \texttt{int} \rightarrow t_3$
$t_f = t_x \rightarrow t_4$

Substitution
$t_0 = t_f \rightarrow (t_x \rightarrow \texttt{int})$
$t_1 = t_x \rightarrow \texttt{int}$
$t_3 = \texttt{int}$
$t_4 = \texttt{int}$
$t_2 = \texttt{int}$

Type Inference: an example (cont'd)

Now, the next equation to be considered contains t_3, which is already bound to `int` in the substitution. So we substitute `int` for t_3 in the equation. We would do the same thing for any other type variables in the equation. We call this *applying* the substitution to the equation.

Equations	**Substitution**
$t_f = \texttt{int} \rightarrow \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow \texttt{int})$
$t_f = t_x \rightarrow t_4$	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

We move the resulting equation into the substitution and update the substitution as necessary.

Equations	**Substitution**
$t_f = t_x \rightarrow t_4$	$t_0 = (\texttt{int} \rightarrow \texttt{int}) \rightarrow (t_x \rightarrow \texttt{int})$
	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \rightarrow \texttt{int}$

Type Inference: an example (cont'd)

The next equation, $t_f = t_x \rightarrow t_4$, contains t_f and t_4, which are bound in the substitution, so we apply the substitution to this equation. This gets

Equations	Substitution
$\texttt{int} \rightarrow \texttt{int} = t_x \rightarrow \texttt{int}$	$t_0 = (\texttt{int} \rightarrow \texttt{int}) \rightarrow (t_x \rightarrow \texttt{int})$
	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \rightarrow \texttt{int}$

If neither side of the equation is a variable, we can simplify, yielding two new equations.

Equations	Substitution
$\texttt{int} = t_x$	$t_0 = (\texttt{int} \rightarrow \texttt{int}) \rightarrow (t_x \rightarrow \texttt{int})$
$\texttt{int} = \texttt{int}$	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \rightarrow \texttt{int}$

Type Inference: an example (cont'd)

We can process these as usual: we switch the sides of the first equation, add it to the substitution, and update the substitution, as we did before.

Equations	Substitution
$\texttt{int} = \texttt{int}$	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

The final equation, $\texttt{int} = \texttt{int}$, is always true, so we can discard it.

Equations	Substitution
	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

We have no more equations, so we are done. We conclude from this calculation that our original expression `proc (f) proc (x) -((f 3),(f x))` should be assigned the type

$$((\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}))$$

Polymorphic types: an example

Let us consider another example: `proc(f)(f 11)`. Again, we start by assigning type variables:

Expression	Type Variable
`f`	t_f
`proc(f)(f 11)`	t_0
`(f 11)`	t_1

Next we write down the equations

Expression	Equations
`proc(f)(f 11)`	$t_0 = t_f \rightarrow t_1$
`(f 11)`	$t_f = \texttt{int} \rightarrow t_1$

And next we solve:

Equations	Substitution
$t_0 = t_f \rightarrow t_1$	
$t_f = \texttt{int} \rightarrow t_1$	

Equations	Substitution
$t_f = \texttt{int} \rightarrow t_1$	$t_0 = t_f \rightarrow t_1$

Equations	Substitution
	$t_0 = (\texttt{int} \rightarrow t_1) \rightarrow t_1$
	$t_f = \texttt{int} \rightarrow t_1$

Type inference: a bad example

```
if x then -(x,1) else 0
```

Expression	Type Variable
`x`	t_x
`if x then -(x,1) else 0`	t_0
`-(x,1)`	t_1

We then generate the equations

Expression	Equations
`if x then -(x,1) else 0`	t_x = bool
	t_1 = t_0
	int = t_0
`-(x,1)`	t_x = int
	t_1 = int

Equations	Substitution
$t_x = \texttt{bool}$	
$t_1 = t_0$	
$\texttt{int} = t_0$	
$t_x = \texttt{int}$	
$t_1 = \texttt{int}$	

Equations	Substitution
$t_1 = t_0$	$t_x = \texttt{bool}$
$\texttt{int} = t_0$	
$t_x = \texttt{int}$	
$t_1 = \texttt{int}$	

Equations	Substitution
$\texttt{int} = t_0$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

Equations	Substitution
$t_0 = \texttt{int}$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

Equations	Substitution
$t_x = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = \texttt{int}$
	$t_0 = \texttt{int}$

Equations
$\texttt{bool} = \texttt{int}$
$t_1 = \texttt{int}$

Substitution
$t_x = \texttt{bool}$
$t_1 = \texttt{int}$
$t_0 = \texttt{int}$

Occurrence Checking

```
proc (f) zero?((f f))
```

Expression	Type Variable
`proc (f) zero?((f f))`	t_0
`f`	t_f
`zero?((f f))`	t_1
`(f f)`	t_2

Expression	Equations
`proc (f) zero?((f f))`	$t_0 = t_f \rightarrow t_1$
`zero?((f f))`	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$
`(f f)`	$t_f = t_f \rightarrow t_2$

Equations
$t_f = t_f \rightarrow \texttt{int}$

Substitution
$t_0 = t_f \rightarrow \texttt{bool}$
$t_1 = \texttt{bool}$
$t_2 = \texttt{int}$

So if we ever deduce an equation of the form $tv = t$ where the type variable tv occurs in the type t, we must again conclude that there is no solution to the original equations. This extra condition is called the *occurrence check*.

This condition also means that the substitutions we build will satisfy the following invariant:

The no-occurrence invariant

No variable bound in the substitution occurs in any of the right-hand sides of the substitution.

Our code for solving equations will depend critically on this invariant.

Type Inference: Implementation

Extending the definition of Type Expressions:

We represent type variables as an additional variant of the `type` data type. We do this using the same technique that we used for lexical addresses in section 3.7. We add to the grammar the production

Type ::= `%tvar-type` *Number*

```
tvar-type (serial-number)
```

Substitutions: basic operation

apply-one-subst : *Type* × *Tvar* × *Type* → *Type*

```
(define apply-one-subst
  (lambda (ty0 tvar ty1)
    (cases type ty0
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (arg-type result-type)
        (proc-type
          (apply-one-subst arg-type tvar ty1)
          (apply-one-subst result-type tvar ty1)))
      (tvar-type (sn)
        (if (equal? ty0 tvar) ty1 ty0)))))
```

This procedure deals with substituting for a single type variable. It doesn't deal with full-fledged substitutions like those we had in the preceding section.

Substitutions as a data type (observer)

A substitution is a list of equations between type variables and types. Equivalently, we can think of this list as a function from type variables to types. We say a type variable is *bound* in the substitution if and only if it occurs on the left-hand side of one of the equations in the substitution.

We represent a substitution as a list of pairs `(type variable . type)`. The basic observer for substitutions is `apply-subst-to-type`. This walks through the type t, replacing each type variable by its binding in the substitution σ. If a variable is not bound in the substitution, then it is left unchanged. We write $t\sigma$ for the resulting type.

The implementation uses the Scheme procedure `assoc` to look up the type variable in the substitution. `assoc` returns either the matching (type variable, type) pair or `#f` if the given type variable is not the `car` of any pair in the list. We write

apply-subst-to-type : *Type* × *Subst* → *Type*

```
(define apply-subst-to-type
  (lambda (ty subst)
    (cases type ty
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (t1 t2)
        (proc-type
          (apply-subst-to-type t1 subst)
          (apply-subst-to-type t2 subst)))
      (tvar-type (sn)
        (let ((tmp (assoc ty subst)))
          (if tmp
            (cdr tmp)
            ty))))))
```

Substitutions as a data type (constructor)

empty-subst : () $\rightarrow$ *Subst*

```
(define empty-subst (lambda () '()))
```

extend-subst : *Subst* $\times$ *Tvar* $\times$ *Type* $\rightarrow$ *Subst*

usage: tvar not already bound in subst.

```
(define extend-subst
  (lambda (subst tvar ty)
    (cons
      (cons tvar ty)
      (map
        (lambda (p)
          (let ((oldlhs (car p))
                (oldrhs (cdr p)))
            (cons
              oldlhs
              (apply-one-subst oldrhs tvar ty))))
        subst))))
```

$$\begin{pmatrix} tv_1 = t_1 \\ \vdots \\ tv_n = t_n \end{pmatrix} [tv = t] \quad = \quad \begin{pmatrix} tv \ = t \\ tv_1 = t_1[tv = t] \\ \vdots \\ tv_n = t_n[tv = t] \end{pmatrix}$$

Unifier

The main procedure of the unifier is `unifier`. The unifier performs one step of the inference procedure outlined above: It takes two types, t_1 and t_2, a substitution σ that satisfies the no-occurrence invariant, and an expression exp. It returns the substitution that results from adding $t_1 = t_2$ to σ. This will be the smallest extension of σ that unifies $t_1\sigma$ and $t_2\sigma$. This substitution will still satisfy the no-occurrence invariant. If adding $t_1 = t_2$ yields an inconsistency or violates the no-occurrence invariant, then the unifier reports an error, and blames the expression `exp`. This is typically the expression that gave rise to the equation $t_1 = t_2$.

Unifier: algorithm

- First, as we did above, we apply the substitution to each of the types t_1 and t_2.
- If the resulting types are the same, we return immediately. This corresponds to the step of deleting a trivial equation above.
- If `ty1` is an unknown type, then the no-occurrence invariant tells us that it is not bound in the substitution. Hence it must be unbound, so we propose to add $t_1 = t_2$ to the substitution. But we need to perform the occurrence check, so that the no-occurrence invariant is preserved. The call `(no-occurrence?` *tv t*`)` returns `#t` if and only if there is no occurrence of the type variable *tv* in *t* (figure 7.5).
- If t_2 is an unknown type, we do the same thing, reversing the roles of t_1 and t_2.
- If neither t_1 nor t_2 is a type variable, then we can analyze further.

  If they are both `proc` types, then we simplify by equating the argument types, and then equating the result types in the resulting substitution.

  Otherwise, either one of t_1 and t_2 is `int` and the other is `bool`, or one is a `proc` type and the other is `int` or `bool`. In any of these cases, there is no solution to the equation, so an error is reported.

Unifier: code

unifier : *Type* × *Type* × *Subst* × *Exp* → *Subst*

```
(define unifier
  (lambda (ty1 ty2 subst exp)
    (let ((ty1 (apply-subst-to-type ty1 subst))
          (ty2 (apply-subst-to-type ty2 subst)))
      (cond
        ((equal? ty1 ty2) subst)
        ((tvar-type? ty1)
         (if (no-occurrence? ty1 ty2)
           (extend-subst subst ty1 ty2)
           (report-no-occurrence-violation ty1 ty2 exp)))
        ((tvar-type? ty2)
         (if (no-occurrence? ty2 ty1)
           (extend-subst subst ty2 ty1)
           (report-no-occurrence-violation ty2 ty1 exp)))
        ((and (proc-type? ty1) (proc-type? ty2))
         (let ((subst (unifier
                       (proc-type->arg-type ty1)
                       (proc-type->arg-type ty2)
                       subst exp)))
           (let ((subst (unifier
                         (proc-type->result-type ty1)
                         (proc-type->result-type ty2)
                         subst exp)))
             subst)))
        (else (report-unification-failure ty1 ty2 exp))))))
```

Figure 7.4 The unifier

No-Occurrence Checking

no-occurrence? : *Tvar* × *Type* → *Bool*

```
(define no-occurrence?
  (lambda (tvar ty)
    (cases type ty
      (int-type () #t)
      (bool-type () #t)
      (proc-type (arg-type result-type)
        (and
          (no-occurrence? tvar arg-type)
          (no-occurrence? tvar result-type)))
      (tvar-type (serial-number) (not (equal? tvar ty))))))
```

Figure 7.5 The occurrence check

Finding type of an expression

optype->type : *OptionalType* → *Type*

```
(define otype->type
  (lambda (otype)
    (cases optional-type otype
      (no-type () (fresh-tvar-type))
      (a-type (ty) ty))))
```

fresh-tvar-type : () → *Type*

```
(define fresh-tvar-type
  (let ((sn 0))
    (lambda ()
      (set! sn (+ sn 1))
      (tvar-type sn))))
```

type-to-external-form : *Type* → *List*

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type)))
      (tvar-type (serial-number)
        (string->symbol
          (string-append
            "ty"
            (number->string serial-number)))))))
```

Type-of

Now we can write `type-of`. It takes an expression, a type environment mapping program variables to type expressions, and a substitution satisfying the no-occurrence invariant, and it returns a type and a new no-occurrence substitution.

The type environment associates a type expression with each program variable. The substitution explains the meaning of each type variable in the type expressions. We use the metaphor of a substitution as a *store*, and a type variable as *reference* into that store. Therefore, `type-of` returns two values: a type expression, and a substitution in which to interpret the type variables in that expression. We implement this as we did in exercise 4.12, by defining a new data type that contains the two values, and using that as the return value.

The definition of `type-of` is shown in figures 7.6–7.8. For each kind of expression, we recur on the subexpressions, passing along the solution so far in the substitution argument. Then we generate the equations for the current expression, according to the specification, and record these in the substitution by calling `unifier`.

Answer = *Type* × *Subst*

```
(define-datatype answer answer?
  (an-answer
    (ty type?)
    (subst substitution?)))
```

type-of-program : *Program* → *Type*

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cases answer (type-of exp1
                        (init-tenv) (empty-subst))
          (an-answer (ty subst)
            (apply-subst-to-type ty subst)))))))
```

type-of : *Exp* × *Tenv* × *Subst* → *Answer*

```
(define type-of
  (lambda (exp tenv subst)
    (cases expression exp

      (const-exp (num) (an-answer (int-type) subst))
```

(zero?-exp e_1)	:	$t_{e_1} = \mathbf{int}$
		$t_{(\texttt{zero?-exp}\ e_1)} = \mathbf{bool}$

```
      (zero?-exp (exp1)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst1)
            (let ((subst2
                    (unifier ty1 (int-type) subst1 exp)))
              (an-answer (bool-type) subst2)))))
```

(diff-exp e_1 e_2)	:	$t_{e_1} = \mathbf{int}$
		$t_{e_2} = \mathbf{int}$
		$t_{(\texttt{diff-exp}\ e_1\ e_2)} = \mathbf{int}$

```
      (diff-exp (exp1 exp2)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst1)
            (let ((subst1
                    (unifier ty1 (int-type) subst1 exp1)))
              (cases answer (type-of exp2 tenv subst1)
                (an-answer (ty2 subst2)
                  (let ((subst2
                          (unifier ty2 (int-type)
                            subst2 exp2)))
                    (an-answer (int-type) subst2))))))))
```

(if-exp e_1 e_2 e_3)	:	$t_{e_1} = \mathbf{bool}$
		$t_{e_2} = t_{(\texttt{if-exp}\ e_1\ e_2\ e_3)}$
		$t_{e_3} = t_{(\texttt{if-exp}\ e_1\ e_2\ e_3)}$

```
      (if-exp (exp1 exp2 exp3)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst)
            (let ((subst
                    (unifier ty1 (bool-type) subst exp1)))
              (cases answer (type-of exp2 tenv subst)
                (an-answer (ty2 subst)
                  (cases answer (type-of exp3 tenv subst)
                    (an-answer (ty3 subst)
                      (let ((subst
                              (unifier ty2 ty3 subst exp)))
                        (an-answer ty2 subst))))))))))

      (var-exp (var)
        (an-answer (apply-tenv tenv var) subst))

      (let-exp (var exp1 body)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (exp1-type subst)
            (type-of body
              (extend-tenv var exp1-type tenv)
              subst))))
```

$$\texttt{(proc-exp}\ var\ body\texttt{)} \quad : \quad t_{\texttt{(proc-exp}\ var\ body\texttt{)}} = (t_{var} \rightarrow t_{body})$$

```
(proc-exp (var otype body)
  (let ((var-type (otype->type otype)))
    (cases answer (type-of body
                    (extend-tenv var var-type tenv)
                    subst)
      (an-answer (body-type subst)
        (an-answer
          (proc-type var-type body-type)
          subst)))))
```

$$\texttt{(call-exp}\ rator\ rand\texttt{)} \quad : \quad t_{rator} = (t_{rand} \rightarrow t_{\texttt{(call-exp}\ rator\ rand\texttt{)}})$$

```
(call-exp (rator rand)
  (let ((result-type (fresh-tvar-type)))
    (cases answer (type-of rator tenv subst)
      (an-answer (rator-type subst)
        (cases answer (type-of rand tenv subst)
          (an-answer (rand-type subst)
            (let ((subst
                    (unifier
                      rator-type
                      (proc-type
                        rand-type result-type)
                      subst
                      exp)))
              (an-answer result-type subst)))))))
```

$$\texttt{letrec}\ t_{proc-result}\ p\ (var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body} \quad :$$
$$t_p = t_{var} \rightarrow t_{e_{proc\text{-}body}}$$
$$t_{e_{letrec\text{-}body}} = t_{\texttt{letrec}\ t_{proc-result}\ p\ (var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}}$$

```
(letrec-exp (p-result-otype p-name b-var b-var-otype
             p-body letrec-body)
  (let ((p-result-type (otype->type p-result-otype))
        (p-var-type (otype->type b-var-otype)))
    (let ((tenv-for-letrec-body
            (extend-tenv p-name
              (proc-type p-var-type p-result-type)
              tenv)))
      (cases answer (type-of p-body
                      (extend-tenv b-var p-var-type
                        tenv-for-letrec-body)
                      subst)
        (an-answer (p-body-type subst)
          (let ((subst
                  (unifier p-body-type p-result-type
                    subst p-body)))
            (type-of letrec-body
              tenv-for-letrec-body
              subst)))))))))
```